ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

Limitations and Opportunities of Modern Hardware Isolation Mechanisms

Xiangdong Chen[1], Zhaofeng Li[1], Tirth Jain[†2], Vikram Narayanan[1], and Anton Burtsev[1]

[1]University of Utah, [2]Maya Labs

bstract

A surge in the 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 of understanding their suitability for the isolation of subsystems with the tightest performance budgets. Our analysis shows that while a huge step forward, the isolation mechanisms 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 [17, 24–26, 63, 64, 80, 87, 99, 102–104, 111, 113, 118, 118, 119], until recently the performance of hardware isolation primitives remained a low priority in commodity CPUs. For example, on x86 machines, one of the low-overhead isolation mechanisms, segmentation [61, 66, 124], was deprecated as part of the transition from 32-bit to 64-bit addressing mode, leaving page tables as the only available isolation mechanism. For decades, lack of hardware support was making fine-grained isolation prohibitively expensive. For example, a highly optimized page-based isolation implementation requires 771 cycles for a cross-subsystem invocation on Intel CPUs (818 cycles on ARM) [1] – an overhead prohibitive for isolation of modern I/O intensive workloads like network processing frameworks [19] which spend less than a hundred cycles per I/O request [30].

The interest in isolation was revived by a surge in complexity and automation of targeted security attacks [53, 100, 107]. Memory Protection Keys (MPK) and Extended Page-Table (EPT) switching with VM functions (VMFUNC) [52] deployed in modern Intel CPUs provide support for memory isolation with overheads gradually approaching the overhead of a function call [41, 76, 83, 112]. 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 [67] and subsystem isolation [75]. ARM Morello is the first silicon implementation of the CHERI capability model [119].

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, 78, 84, 124], database extensions [15, 105], virtualized network functions [6, 45, 48, 73, 92, 96], web applications [3, 27, 32, 56], serverless cloud and edge platforms [4, 5, 58, 90, 109], and operating system kernels [75, 81, 83].

As the range of new hardware mechanisms matures the questions of their practicality and relative advantages for isolation arise. The overhead of switching the isolation boundary is becoming progressively lower, e.g., writing a `pkru` register that changes the current tag in the Intel MPK isolation mechanism introduces an overhead of only 20-26 cycles [41, 89]. At the same time, new isolation mechanisms often rely on a complex compiler and binary instrumentation to enforce isolation [41, 75, 112], and require software and hardware schemes to support the revocation of rights [117, 120].

Our work performs an in-depth look at hardware isolation mechanisms deployed in recent ARM and x86 CPUs with the goal to understand their ability to support fine-grained isolation of subsystems with the strictest performance budgets. We first introduce a generic isolation scheme aimed at providing low-overhead isolation and efficient zero-copy communication. We draw inspiration from prior projects aimed at implementing fine-grained, process-like boundaries around untrusted subsystems [47, 82]. We then develop carefully optimized implementations of the isolation primitives for dif-

[†]Work done at the University of Utah

ferent CPU architectures: Intel MPK, ARM MTE, ARM PAC, and ARM Morello. These four mechanisms support isolation with a low overhead and fast switching of isolation boundary. A controlled implementation allows us to reason about the benefits and limitations of each hardware mechanism.

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 the 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 the 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 the passing of zero-copied memory regions across all cores of the system, which results in a restrictive programming model (the buffers passed on one core cannot be accessed from other cores). Tag-based schemes like MPK and MTE support an impractically small 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 support for revocation of rights and "move" semantics, i.e., ensuring that the caller loses 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, CHERI, and even PAC 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 minimize the use of costly compiler instrumentation which becomes prohibitive in modern systems (Intel MPK and ARM Morello satisfy this principle, but ARM MTE and AMR PAC does not). ***ore-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 the isolation boundary are available to all threads and even more important for the global revocation of rights (our analysis shows that ARM MTE is the only set of isolation extensions that implements this principle). ***Number of isolated subsystems***: Hardware isolation primitives should support practically large number of isolated subsystems. ***Revocation***: Support for revocation should be a first-class design principle. In the face of frequent communication, the ability to revoke access rights to a the specific memory region is as important as granting them.

We demonstrate that the above principles are largely overlooked by the current generation of hardware isolation mechanisms. Arguably, this might be due to the lack of understanding of which functionality is needed for implementing isolation. The goal of this work, therefore, is to improve the understanding of modern isolation mechanisms, analyze their advantages and disadvantages, and, hopefully, shape the design space of the future solutions. We argue that in most cases existing mechanisms can be extended to support the above principles (although some changes, e.g., support of revocation in pointer-centric architectures like CHERI, remain challenging research problems).

2 Background

The very first isolation mechanisms root back to the early time-sharing machines that utilized hardware segmentation and later paging to support efficient virtualization of memory. The first computer architectures (1950-1960) were batch machines which executed one program at a time. The need for multi-programming required dynamic sharing of memory across multiple programs and triggered development of the first isolation mechanisms. In 1961, the Burroughs B5000 introduced segmentation [87] (similar ideas appear in the Rice University Computer [51, 55], and the Basic Language Machine [49, 50] architectures). Paging was introduced by the Atlas Computer [59] but naturally as a storage management mechanism not a security primitive.

In 1977, the final Multics report suggested that the core kernel can be reduced to several thousands of lines of code and even formally verified while the rest of the operating system can execute in isolation [68]. Since then, for over four decades, a range of user-level and kernel projects explored the possibility of providing practical, low-overhead isolation through 1) hardware mechanisms that were used to isolate kernel subsystems [38, 81, 83, 106], support virtualized execution [13, 37, 57, 85], and microkernelization [11, 21, 22, 34, 42–44]; 2) programming language safety [9, 12, 47, 82, 115]; and 3) software fault isolation (SFI) [14, 31, 35, 72, 74, 91, 94, 116, 126, 127]. In parallel, numerous hardware architectures explored ideas of fine-grained isolation centered around memory tags and capabilities [17, 24–26, 63, 64, 80, 87, 99, 102–104, 111, 113, 118, 118, 119].

Unfortunately, commodity CPU architectures remained optimized for the abstraction of process which was central to time-sharing, i.e., designed to execute a collection of largely independent programs with infrequent inter-process communication. 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. Efficient switching between isolated subsystems was never a goal [29]. Ironically, the x86 architecture abandoned segmentation (which was demonstrated as an efficient isolation mechanism by both microkernel [66] and software fault iso-

lation [61, 124] architectures) while transitioning to 64-bit addressing mode.

Trying to remove hardware overheads from 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, 14, 31, 72, 95, 116, 124]. 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 [116]. 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 [27, 32, 56], data streaming platforms [5], serverless and edge platforms [4, 58, 90, 109] as well as providing practical isolation of browser [2, 78, 84, 124] and kernel [14, 31, 72, 98] extensions implemented in unsafe languages that are prone to low-level memory safety violations and vulnerabilities. Unfortunately, as we argue in this paper, without proper hardware support the overheads of SFI remain high (this is consistent with findings by Narayan et al. [79] and by Yedidia [123]).

2.1 Modern Isolation Primitives

In the last decade, commodity CPUs introduced a diverse range of hardware primitives aimed at supporting 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 [83], SPP controls only write accesses which is insufficient for enforcing confidentiality). Other mechanisms, however, bear the promise of improving the 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, a fast operation taking 20-26 cycles [41, 89].

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 [112] or dynamic validation of all `wrpkru` instructions with hardware breakpoints [41]. 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].

RM 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 the 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 does not directly support isolation – the attacker can change the upper bits of the pointer containing 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.

RM P C 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, allows 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 [67], spatial and temporal [33, 65] safety and isolation of subsystems [75].

RM Morello ARM Morello is an experimental architecture that implements the CHERI capability model [39, 119]. 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 the 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. IBT marks the jump targets in a program with a special instruction (`endbr64`). On an indirect call, the hardware enters the `WAIT_FOR_ENDBRANCH` state and generates an exception if the next encountered instruction is not `endbr64`.

3 Design principles for efficient isolation

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, 45, 48, 73, 88, 92, 96], loadable kernel extensions and device drivers [14, 46, 72, 81, 83, 106], etc. The isolation architecture protects the state of each subsystem, i.e., its heap, stacks, etc., from accidental and malicious accesses by other subsystems, and provides a way for controlled communication in which subsystems can invoke each other interfaces.

Low-overhead enforcement To be practical, isolation 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, are 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 5).

Fast switching of isolation boundaries Fine-grained isolation comes at the cost of frequent crossings of isolation boundaries. Historically, overheads of cross-subsystem invocations remained high [60]. A typical invocation required a transition into a privileged execution mode to 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 the 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 [23, 72, 81, 83], network processing frameworks [88], databases [62], etc., requires frequent passing of data between isolated subsystems. In such systems, the overhead of copying data between subsystems is prohibitive. An isolation mechanism should support low-overhead zero-copy passing of data across 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 the caller to the 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 has been passed to the callee. 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 [117, 120] or, as we demonstrate in this work, compiler instrumentation to track the propagation of all capabilities in memory. Second, revocation should be en-

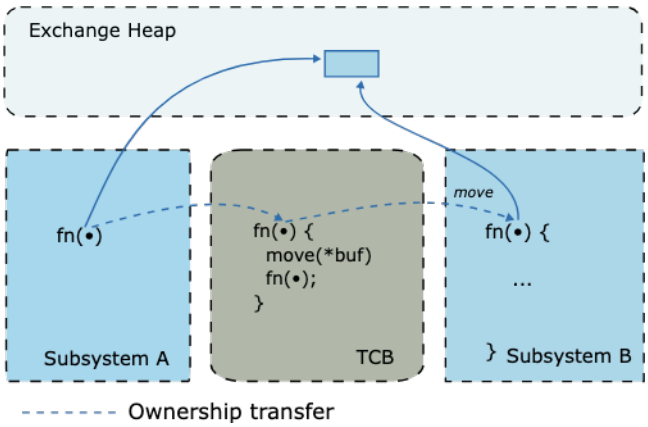

Figure 1: Private and shared heaps

forced across the cores. This is challenging as cores can have capabilities loaded in registers. As a result, synchronization of access rights requires an expensive inter-processor interrupt (IPI) or alternatively a restricted programming model in which the processing of a specific object is pinned to one CPU core.

Fast, zero-copy isolation scheme In the sections below we implement the following isolation scheme (Figure 1). To fully leverage the low-overhead nature of the hardware mechanism, we implement a *migrating threads* model of invocation [36] that avoids overheads of dispatching messages 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 other's private heaps [47, 82]. Specifically, we orchestrate isolated subsystems as a collection of isolated private heaps and a special shared *exchange heap* – a heap that allows the 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 approach helps us avoid a scenario where objects left in an inconsistent state by a crashing subsystem become accessible on the shared heap by other subsystems.

4 Isolation with Modern Mechanisms

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

4.1 Intel MPK + CET

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 edge. MPK does not check memory tags for targets of indirect control flow transitions. An attacker can find a valid (with respect to IBT) entry point in another subsystem and potentially reach code that contains sensitive instructions (e.g., attack gadgets, trampolines to other subsystems, etc.). To prevent indirect control transitions between subsystems, we instrument indirect control flow instructions (e.g., jumps and calls) with a memory load from the target address of the control flow transition, hence using MPK to validate that the address belongs to the same subsystem.

To implement support for zero-copy communication, we support the 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 updating 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 multi-threaded 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. A 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. On our Intel Core i7 machines, a `wrpkru` instruction takes around 40 cycles. To avoid the overhead of additional `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 use `assert` syntax instead of complete machine code to simplify the text). 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 return from the callee, 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.

```
; check buffer ownership before entering B
rdpkru ; rax contains permissions of
and rax, r10 ; r10 has the buffer(s)   wants to pass to B
assert rax == 0 ;   has the permissions of the buffer(s)
call .reserve_ssp ; decrement ssp and rsp by 8 bytes
reserve_ssp:
  pop ; restore rsp
  rdsspq r9 ; copy ssp to r9
  wrssq [r9], rax ; save pkru on the top of the shadow stack
  ; grant buffer ownership to B
  mov rax, PKRU_B
  xor r10, 0xFFFFFFFFFFFFFFFF
  and rax, r10
  ; switch to B
  wrpkru
  ; locate stack
  ...
  call dispatch
  ...
  ; return path
  ; load pkru of caller from the shadow stack
  rdsspq r10
  mov rax, [r10]
  mov r10, 0x1
  incsspq r10 ; increment ssp by 1 x 8 bytes.
  ; switch back into
  wrpkru
  ; restore the stack, restore register state...
```

Listing 1: MPK trampoline between two subsystems, and B

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) [41, 112]. While plausible for simple isolation scenarios (one isolated subsystem and no zero-copy passing of buffers), MPK alone becomes an easy target for a variety of attacks without 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 to B, and then to C an attacker can return from C directly into , and even return into a random subsystem D). CFI not only eliminates the complex binary rewriting required to protect MPK sandboxes from control flow attacks [18, 41, 112], but also allows for semantically complex transfer of rights on cross-subsystem invocations.

4.2 RM MTE

To enforce isolation with MTE, we tag the 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 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. Similar to MPK, 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 [95]. 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 the 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 the entire private heap with the subsystem's MTE key. Objects allocated on the shared exchange heap are also tagged with the MTE key of the subsystem that can access them at the moment (note, we enforce single ownership on the shared heap and move the objects between subsystems upon invocations). 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 retagging it with the tag of the callee. Note that since the tag metadata is updated in memory, the access rights are reflected on all CPUs. Combined with single ownership, MTE provides support for reclamation of resources allocated on the shared heap.

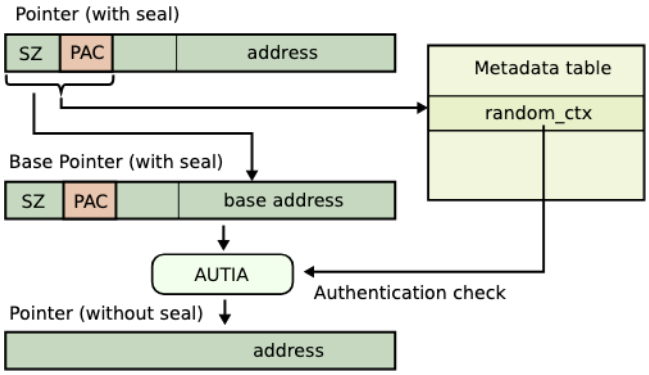

Figure 2: Enforcement with PAC

4.3 RM P C

ARM PAC provides a way to authenticate individual memory pointers, i.e., implement a check that the pointer was not made up or altered by the isolated subsystem but was obtained from the TCB (e.g., from the memory allocator) or was received through a valid communication channel. Combined with authentication, an in-memory metadata can be used to check for spatial bounds of the object accessed via a pointer, its liveness and access permissions [33, 65, 75]. In a way, PAC allows implementing CHERI-like capability scheme but relying on

a single hardware primitive – signing and authentication of pointers. Similar to other SFI schemes, PAC requires careful instrumentation of every memory instruction to ensure that access is allowed.

Specifically, we allocate all shared and private heap objects with an alignment that matches the size of the object rounded 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. this allows us to represent objects of size up to 2^{255} bytes. We use the size information to truncate the lower bits of the pointer to restore the base address of the object allocated on the heap (e.g., the pointer can point into the middle of an object). We use the following signing context: the base address bits of the pointer along with the size bits in the upper bits, and a random identifier which we generate when the object is allocated. To ensure the liveness of the object, we record the random identifier in a global metadata table indexed by the top 16 bits of the pointer (size bits and PAC signature). If we detect collision in the table we pick a new random identifier to generate a new PAC signature that indexes into a new entry. Before each load and store, we first look up its random identifier in the global metadata table using the top 16 bits . We authenticate the sealed (i.e., signed) base pointer using the random identifier. Authentication fails if the PAC signature, size bits, or the address itself is changed. Then, we remove the PAC from the pointer and perform the memory operation safely. If the pointer participates in a pointer arithmetic, we do not change the PAC signature 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 an object is deallocated, the metadata is updated to store an invalid random identifier. If the object is moved to another subsystem in a cross-subsystem invocation, the pointer is re-signed with a new random identifier. The old random identifier is cleared from the metadata table (this effectively revokes all aliases that might remain in the caller subsystem, making them invalid).

4.4 RM Morello

The ARM Morello architecture implements hardware bounds checks on every pointer dereference, i.e., on every load and store instruction. Hence, Morello allows one to implement an isolation scheme similar to the one we discussed above for ARM PAC but entirely in hardware (hence with virtually no overhead). The architecture ensures that it is impossible to make up capability pointers (they can only be obtained from TCB), and enforces a bounds check on every memory access. While appealing such straightforward isolation scheme has a flaw in practice. Morello does not provide a way to revoke capabilities and hence fails to support the move semantics on cross-subsystem invocations. Capabilities may be freely duplicated by the isolated subsystem, which can, for example, store capabilities in memory and subsequently load and use them even after moving them to another subsystem.

To support revocation and move semantics, we extend the above isolation scheme with a special instrumentation pass that tracks all capability stores in memory with the goal to invalidate them when they are revoked. The compiler pass inserts calls to a trusted runtime which records the fact that capability has been stored in memory. When capability is moved into another subsystem, we leverage recorded information to invalidate (hence revoke) all copies of the moved capability in memory.

Note that if we use the full capability mode, i.e., all memory accesses involve capability pointers, the instrumentation becomes prohibitively expensive as we need to track memory stores for all pointers in the code of the isolated subsystem. We therefore leverage Morello's hybrid mode that allows mixing capability and non-capability (legacy) memory accesses. Similar to MPK and MTE, we separate the memory of isolated subsystems 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. Objects on the private heap are accessed through regular, non-capability memory accesses (Morello uses the detaulf data capability (DDC) to check the bounds of the address space). Objects on the shared heap are accessed with explicit capability loads and stores. This allows us to instrument only the store instructions for the capability registers. Most loads and stores of subsystem-local data are integer operations and therefore not subject to instrumentation.

To transition between the executive and restricted modes, the sealed function capabilities are used for both the forward and return edges.

Finally, capabilities in Morello can remain resident in registers, meaning that a thread may retain access to a region of memory even after the capability to this region is moved to another subsystem by another thread. To restrict this behavior, we associate each capability with a *single* thread that can use it. Specifically, in each capability, the user-defined permission bits are used to store the identifier of the owning thread. We store the current thread identifier as a sealed capability in the Compartment ID (`cid`) register, which is a capability register whose semantics are software-defined. Like capability stores, we also instrument capability loads to ensure each thread can only load capabilities that it owns.

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 for 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

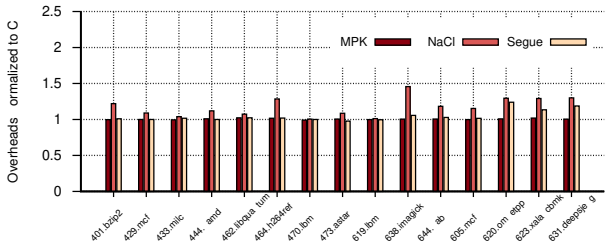

Figure 3: SPEC CPU 2006 and SPEC CPU 2017 (x86)

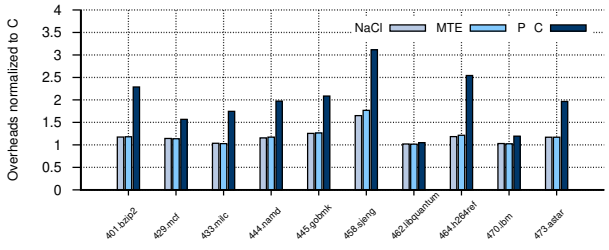

Figure 4: SPEC CPU 2006 and SPEC CPU 2017 (ARM)

trampoline switches to Executive mode and loads the DDC of the callee using the Executive DDC. We find that switching between the hardware-supported Executive and Restricted contexts is about 60 cycles faster than switching the DDC manually in the trusted trampoline.

5 nalysis: Limitations and Opportunities

To reason about the performance of different isolation mechanisms, we leverage several hardware platforms. For MPK and CET on 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 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) [40]. 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. For Morello experiments, we use the Morello Hardware Development Platform, which runs a custom SoC based on the Neoverse N1 core [86].

5.1 Overhead of enforcement

To understand the overhead of enforcing isolation, we run a collection of SPEC CPU 2006 and SPEC CPU 2017 benchmarks on Intel (Figure 3), ARM (Figure 4) and Morello (Figure 5) CPUs. As a baseline, we also measure the performance of a purely software SFI isolation scheme simi-

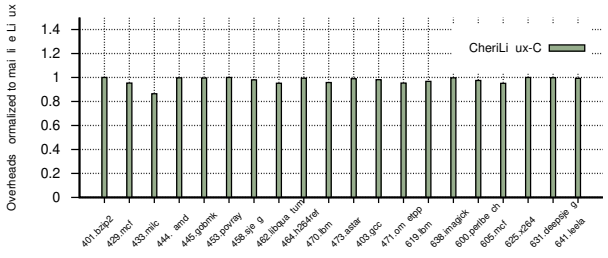

Figure 5: SPEC CPU 2006 and SPEC CPU 2017 (Morello)

lar to Google NaCl [95, 124]. Conceptually NaCl is similar to well-optimized modern SFI implementations, e.g., ones used by Wasm [2]. We implement the NaCl scheme from scratch. Control over implementation allows us to selectively disable individual bits of SFI enforcement, e.g., load and store masking, return address masking, instruction bundling, etc., to reason about the impact of each mechanism on the overall performance of the isolation scheme. 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 slightly faster than WASM).

A high-level observation from our SPEC experiments is that all isolation schemes that rely on compiler instrumentation have high overhead. On x86, NaCl has an average overhead of 17.4%, whereas a combination of MPK and CET has a negligible (0.4%) overhead. On ARM, NaCl-like SFI has an average 18% overhead. As MTE requires two additional reserved registers, it results in 20% average overhead. Due to complex software checks for memory accesses, PAC 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 absolute terms, Morello hardware is significantly slower than the ARM chip used on the Google Phone. On SPEC benchmarks Morello hardware is slower than the ARM chip on the Google Phone from 1.1x to 2.5x, averaging at 1.4x slowdown. Arguably, this is expected as Morello is the first generation of hardware. We assume that significant improvement is possible in the future.

Performance breakdown To get a deeper insight into the reasons for the overheads introduced by the SFI instrumentation, we leverage our purely software NaCl scheme. Several implementation details are important for understanding our analysis. Similar to NaCl, our SFI implementation relies on address masking which is the fastest way of enforcing segment bounds according to our empirical analysis. 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

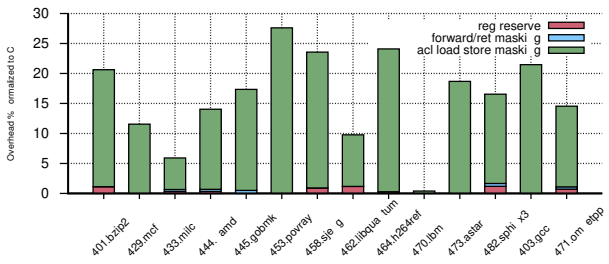

Figure 6: SPEC CPU 2006 performance breakdown (x86)

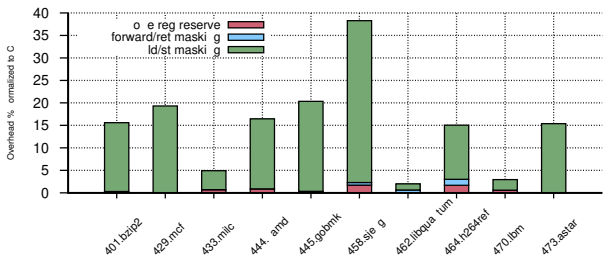

Figure 7: SPEC CPU 2006 performance breakdown (ARM)

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, for memory operations involving two address registers, we use load effective address (`lea`) instead of `mov` to compute the address and 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, #32 ;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 [95, 124]. This allows us to avoid allocating additional registers required to protect the stack. To ensure the integrity of the bounds checks, we mask the indirect control flow transitions and returns from procedures to land at the beginning of a basic instruction block.

To understand the overheads of four mechanisms – 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

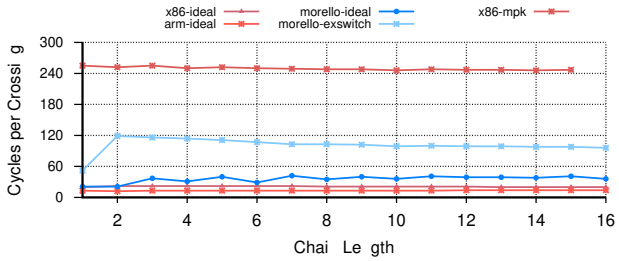

Figure 8: Overhead of cross-subsystem invocations

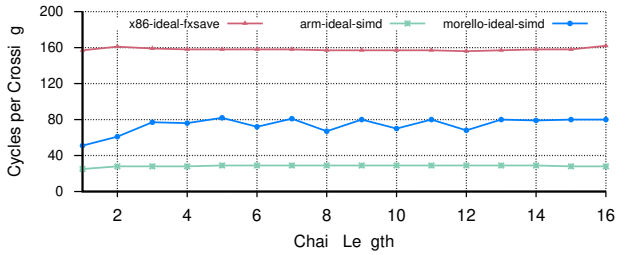

Figure 9: Overhead of cross-subsystem invocations with extended register saving

previous studies [2, 54]. Most of the overhead comes from the added instructions that load the target addresses for loading/storing into 32-bit 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 the 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 the 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 on modern workloads. The masking instruction is on the critical path of the pipeline as the load and store after it depends on it. The limitation above allows us to identify the following design principle:

> **Software transparency** *rchitectural mechanisms should avoid relying on extensive software instrumentation for enforcement of isolation.*

Possible solutions On x86 machines, MPK implements this design principle through a combination of hardware access checks (MPK tags) and control flow enforcement (CET). Together, MPK and CET result in a negligible overhead as only minimal control-flow related instrumentation is required to

enforce isolation.

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 similar to Segue [79] (`segue`, Figure 6). Segue relies on the `gs` segment register to enforce the bounds check on every memory access. By using (`gs`) instead of (`r15`) to keep the base, we 1) free a general purpose register, 2) avoid emitting an extra `mov`/`lea` before a memory access, and 3) free an operand in memory access operation.

```
1    ; load value at [rax + rbx] to rcx
2    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%. Recently, Yedidia implemented a similar scheme for ARM [123].

For ARM MTE, assume that it is possible to keep the tag associated with the heap of the current subsystem in a protected register rather than in each pointer. This tag from the protected register can then be applied in the MTE check against the tag of the accessed memory region Essentially, this eliminates the need for compiler instrumentation, as the tag comes from a protected register rather than a pointer.

To verify the benefits of such an 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 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 for holding the tag.

5.2 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 the hardware boundary switching and the cost of saving and restoring the state of the thread across invocations, 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 (`x86-ideal`, `arm-ideal`, `morello-ideal`).

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 the executive mode turns out to be faster than deriving

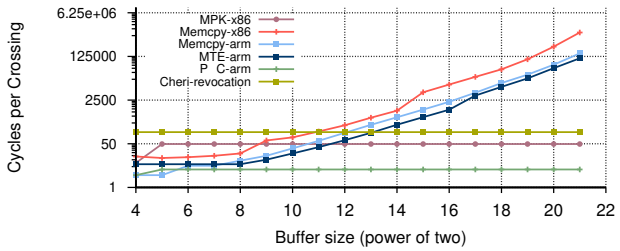

Figure 10: Overhead of passing data

capabilities from the unsealed PCC capability at a difference of about 100 cycles.

Saving and restoring extended registers introduce an overhead of additional 137 cycles, bringing the total cost of a cross-subsystem invocation to 156-162 cycles on Intel (Figure 9, `x86-ideal-fxsave`), 14 extra cycles on ARM (`arm-ideal-simd`), and around 40 extra cycles on Morello (`morello-ideal-simd`). Note that on all architectures, we save and restore the entire FPU/SIMD state and do not take into account any specific calling convention. We observe a greater variance on Morello as we were unable to fix the CPU frequency reliabliy.

Limitations The 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, there are additional optimizations that could be implemented to reduce the overhead of saving both general and extended registers, as well as optimizing the implementation of crossing the isolation boundary. Saving and restoring extended registers constitute the most significant portion of the overhead in a cross-subsystem invocation and could potentially be optimized in hardware.

5.3 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 (i.e., from 16 bytes to 4MB). We evaluate the following IPC mechanisms: 1) transfer of an MPK tagged buffer (`MPK-x86`); 2) memory copy for NaCl-like SFI schemes on Intel and ARM (`memcpy-x86` and `memcpy-arm`); 3) MTE retagging with the `stg` and `dc gva` instructions; 4) moving of a PAC pointer, i.e., upon moving the pointer is re-signed, a new random number is generated and the metadata table is updated (`P C-arm`); 5) passing a capability and revoking one stored capability from memory on Morello CHERI (`Cheri-revocation`).

PAC pointer re-signing shows the lowest overhead at around 5 cycles. We use a fast pseudo-random hash function, FNV-1 (the overhead reaches 50 cycles if we use `rand()` from `libc`). Writing an MPK `pkru` register takes 49 cycles. Passing a CHERI capability that was stored in memory once takes 142 cycles. The overhead of CHERI revocation increases linearly with the number of times the capability is stored in memory. Surprisingly, MTE retagging is only slightly faster than mem-

ory copy. Depending on the size of the object, MTE retagging is 1.2x to 2x faster than memory copy. There are three instructions available for MTE retagging: `stg` (tag 16 bytes), `st2g` (tag 32 bytes), `dc gva` (tag a cache line size of memory). Their performance vary depending on the size of the object. We choose the optimal instruction for each individual object size.

Limitation The main limitation of the MPK scheme is the inability to reflect the passing of zero-copied memory regions across all cores of the system. Upon a cross-subsystem invocation, the `pkru` register is updated on one core to reflect the change in access rights between the caller and callee subsystems. This change, however, is local to the core. Updates of `pkru` registers on other cores require an expensive cross-core synchronization similar to a TLB shootdown, i.e., a traditional inter-processor interrupt (IPI) or an alternative 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 boundaries are available to all threads as well as revocation of rights (which we discuss below).

Possible solutions Implementation of core-coherent zero-copy passing of memory regions for MPK might be possible through a combination of instruction set extensions which could provide controlled access to the tag bits in the page table and support for core-coherent TLBs. Controlled and secure modification of the tag bits stored inside the page table can provide access to the bits themselves but not to the rest of the page table entry. This would allow tag updates in a manner similar to MTE. We can then assign each isolated subsystem a unique tag and update the tag of 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 the TCB, but the risks are high since a compromise of the TCB provides an attacker with unrestricted access to the page table and hence a system-wide control of memory. Implementation of cross-core coherent updates 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. Hardware architects explored support for coherent TLBs in the past [8, 16, 69, 93, 97, 108, 114, 121] which arguably can be brought to modern commodity CPUs.

Limitation Tag-based schemes like MPK and MTE suffer from a limited 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).

> **Number of isolated subsystems:** *Hardware isolation primitives should support practically large number of isolated subsystems.*

Possible solutions Increasing the number of supported tags to a practically large number is challenging. For example, in the 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., the format of the entry and, possibly, the overall layout of the page table.

In the 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).

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

Possible solutions 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 larger tags can support faster passing of data across subsystems. To support different tag sizes in a single process, the tag size information can be kept in unused bits of the page table entry (i.e., each page can have different tag size).

5.4 Revocation

To understand the impact of implementing revocation for the Morello architecture, we develop compiler instrumentation that tracks capability stores in memory and invalidates them on cross-subsystem invocations (Figure 10, `Cheri-revocation`). Specifically, we implement an efficient hash table that tracks memory addresses where each capability is saved and invalidates all capabilities that are moved into another subsystem. On a cross-subsystem invocation, we look up the hash table and invalidate capabilities stored in memory by overwriting them. The overhead of a hash table lookup and invalidation reaches 142 cycles. If the capability is never stored on the heap, an empty hash table lookup takes only 70 cycles.

Limitations Capability schemes like CHERI are inherently limited by the lack of support for revocation. Unrestricted propagation of capabilities requires additional software mechanisms to either scan the private heap of the subsystem for capabilities that need to be revoked [117, 120] or instrumentation that tracks memory locations where capabilities are saved (the approach we suggest in this paper). Both techniques incur significant overhead.

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, CHERI and even PAC (despite the fact that PAC relies on metadata in memory, after metadata is invalidateed, if the pointer is already authenticated, subsequent memory accesses through a pointer are allowed), 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 the 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 a check requires an expensive cross-core synchronization mechanism to preempt execution on all cores, scan registers of all threads (some temporarily saved in memory) and invalidate them.

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

Possible solutions At the moment, MTE is the only solution that leverages a centralized in-memory metadata region and hence implements support for revocation (we assume that the hardware avoids optimizations that eliminate tag checks between accesses to the same memory location). 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 and controlled access to tags in the page table (in a manner similar to core-coherent synchronization of rights discussed above).

Implementing revocation for CHERI capabilities is arguably the most challenging due to distributed nature and unrestricted flow of capabilities. A classical object capability approach is to revoke access rights with proxies [77], 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 an approach, instead of pointing to the memory region, a 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.

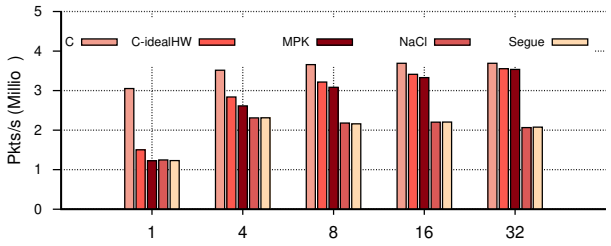

Figure 11: NFs performance on varying batch sizes (x86)

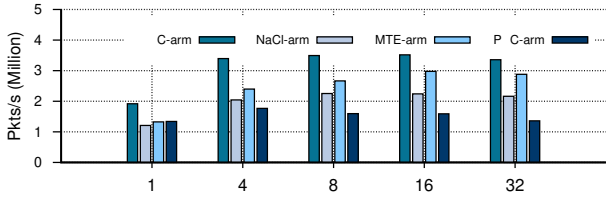

Figure 12: NFs performance on varying batch sizes (ARM)

5.5 End-to-End pplication Use-Cases

To understand a combined impact of enforcing isolation in software, we design several application benchmarks: 1) a network function virtualization framework and 2) a video processing pipeline typical for modern serverless workloads.

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 [88]. Today, a wide range of *network functions* (NFs) handle the most complex network tasks such as intrusion detection, packet filtering, load balancing, etc. NFs often have conflicting reliability and security goals, necessitating isolation [71, 73, 101, 122, 125]. Isolation of NFs remains a challenging problem due to stringent performance requirements of packet processing applications [6, 45, 48, 73, 92, 96]. 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.

We implement four network functions: (1) **TTL** which decrements the time-to-live field in a packet's IPv4 header, (2) **N T** which rewrites the source IP and port of a packet according to a mapping, (3) **CL 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 [28]. We configure Maglev with 65 K backend servers and 1 M flows.

On small batch sizes, the cost of MPK-based isolation impacts the performance of the network function chain (Figure 11, Figure 12 and Figure 13). 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 35% due to frequent buffer copying. Similarly, Although MTE retagging is faster than buffer copying, MTE

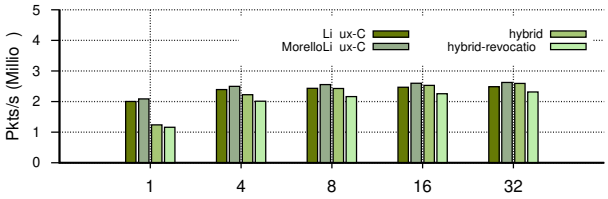

Figure 13: NFs performance on varying batch sizes (Morello)

still incurs around 15% overhead.

CHERI has four different configurations: 1) `Linux-C` runs monolithic NFs with mainline Linux; 2) `MorelloLinux-C` runs monolithic NFs but on top of the CHERI-enabled Linux; 3) `hybrid` executes TCB inside the CHERI executive mode with isolated subsystems runing in restricted mode, network packets are passed on the shared heap using capability pointers; 4) `hybrid-revocation` is the same as above but with insrtumentation required for revocation of capabilities. Overall, CHERI demonstrates good performance in this experiment. Without load/store instrumentation, on batch of 32, hybrid mode has only 1% overhead compared to unmodified C version under the same kernel version. Switching from regular C pointers to capability pointers has negligible overhead. Most overhead comes from domain switching. In all network functions, no instance of capability store is observed, and the capability loads are very infrequent as well. As a result, even with revocation and instrumentation on capability load/store, `hybrid-revocation` has just 11% overhead compared to C.

Video processing Video processing is a workload typical for modern serverless platforms [70] – fast serverless functions extend interactive but relatively inefficient core of the web application. We implement a video processing pipeline that extracts frames from an input video to produce an animated thumbnail (GIF). 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 the FFmpeg library [110]. We then isolate two processing stages with five different isolation mechanisms: NaCl, Segue, MPK, MTE, and PAC (Figure 14, Figure 15). We were unable to evaluate the CHERI configuration as FFmpeg library requires an extensive rewrite to use capability pointers for the data buffers passed between isolated subsystems. Intel MPK and ARM MTE/-PAC leverage benefits of zero-copy, i.e., extracted frames are retagged/resigned and passed between compartments. NaCl and Segue require a memory copy.

The frame extraction takes 33x more CPU time than thumbnaiil processing (e.g., on x86 the frame extraction takes 5.3 billion cycles while thumbnail generation takes only 160 million cycles). We therefore report normalized time to compare overheads of individual stages side by side. Moreover, buffer passing takes less than 0.1% of the total execution time for all the tests (e.g., 5 million out of 5 billion cycles on x86).

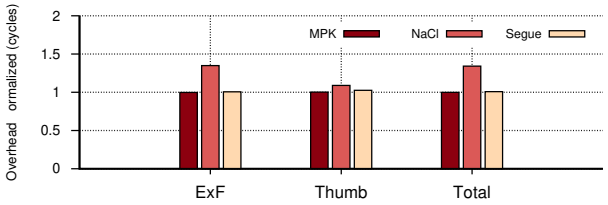

Figure 14: Overhead of FFmpeg (x86)

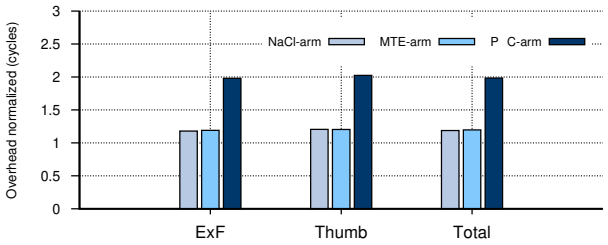

Figure 15: Overhead of FFmpeg (ARM)

Hence, lack of zero-copy does not impact this experiment significantly.

On x86, MPK performs extremely close to non-isolated C with less than 0.3% overhead. Software only SFI schemes, i.e., NaCl, suffer from high overheads of the bounds checks on all memory accesses and from the impact of register reservation (35% overhead on the Frame Extractor and 10% overhead on the Thumbnail Encoder). Interestingly, leveraging the `gs` segment trick, Segue has less than 3% overhead. On ARM, NaCl has an average overhead of 17-20%. Similar to SPEC benchmarks, the overhead of MTE is slightly higher than NaCl and is around 20%. PAC has the worst performance with the overhead reaching 100%.

6 Conclusions

After decades of relatively slow adoption, we finally see a renewed interest in architectural support for isolation. Our work studies advantages and limitations of the recent hardware isolation mechanisms with the goal of deriving a set of design principles critical for achieving practical isolation. Our analysis identifies several problems in recent hardware mechanisms, and suggests approaches to address them. We hope that our work can be useful for development of the next generation of hardware isolation mechanisms.

cknowledgments

We would like to thank OSDI'23, EuroSys'23 and USENIX ATC'24 reviewers as well as our shepherd, Emmett Witchel, for numerous insights helping us to improve this work. This research is supported in part by the National Science Foundation under Grant Numbers 2313412, 2341138 and 2239615.

References

[1] seL4 Performance. https://sel4.systems/ bout/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 CM/IEEE Symposium on rchitectures 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/ rm%20Developer%20Community/PDF/ rm_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 rchitectures and Compilation Techniques (P CT)*, pages 273–287, 2017.

[9] Godmar Back and Wilson C Hsieh. The KaffeOS Java Runtime System. *CM Transactions on Programming Languages and Systems (TOPL S)*, 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] Bomberger, A.C. and Frantz, A.P. and Frantz, W.S. and Hardy, A.C. and Hardy, N. and Landau, C.R. and Shapiro, J.S. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel rchitectures*, pages 95–112, 1992.

[12] 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.

[13] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in Linux. pages 9–22, 2010.

[14] 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 CM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP'09, pages 45–58. ACM, 2009.

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

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

[17] Silviu Chiricescu, André DeHon, Delphine Demange, Suraj Iyer, Aleksey Kliger, Greg Morrisett, Benjamin C Pierce, Howard Reubenstein, Jonathan M Smith, Gregory T Sullivan, et al. SAFE: A clean-slate architecture for secure systems. In *Technologies for Homeland Security (HST), 2013 IEEE International Conference on*, pages 570–576. IEEE, 2013.

[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/IS Future/319433 034.pdf, 2018.

[21] Data61 Trustworthy Systems. *seL4 Reference Manual*, 06 2017. http://sel4.systems/Info/Docs/seL4-manual-latest.pdf.

[22] DDEKit and DDE for linux. http://os.inf.tu-dresden.de/ddekit/.

[23] 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.

[24] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. HardBound: Architectural Support for Spatial Safety of the C Programming Language. *SIG RCH Comput. rchit. News*, 36(1):103–114, mar 2008.

[25] Udit Dhawan, Nikos Vasilakis, Raphael Rubin, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Benjamin C. Pierce, and André DeHon. PUMP: A Programmable Unit for Metadata Processing. In *Proceedings of the Third Workshop on Hardware and rchitectural Support for Security and Privacy*, HASP '14, New York, NY, USA, 2014. Association for Computing Machinery.

[26] 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 rchitecture*, ISCA'19, pages 671–684, New York, NY, USA, 2019.

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

[28] 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.

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

[30] 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 CM/IEEE Symposium on rchitectures for Networking and Communications Systems (NCS)*, pages 1–13. IEEE, 2019.

[31] lfar 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.

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

[33] 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.

[34] Feske, N. and Helmuth, C. *Design of the Bastei OS architecture*. Technische Universität, Dresden, Fakultät Informatik, 2007.

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

[36] 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.

[37] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *In 1st Workshop on Operating System and rchitectural Support for the on demand IT InfraStructure (O - SIS*, 2004.

[38] 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 CM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*, pages 109–114. ACM, 2000.

[39] 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.

[40] 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.

[41] 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 nnual Technical Conference (USENIX TC'19)*, pages 489–504, July 2019.

[42] Heiser, G. and Elphinstone, K. and Kuz, I. and Klein, G. and Petters, S.M. Towards Trustworthy Computing Systems: Taking Microkernels to the Next Level. *CM SIGOPS Operating Systems Review*, 41(4):3–11, 2007.

[43] Herder, J.N. and Bos, H. and Gras, B. and Homburg, P. and Tanenbaum, A.S. MINIX 3: A highly reliable, self-repairing operating system. *CM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[44] Hohmuth, M. and Peter, M. and Härtig, H. and Shapiro, J.S. Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors. In *Proceedings of the 11th workshop on CM SIGOPS European workshop*, page 22. ACM, 2004.

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

[46] 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.

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

[48] 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.

[49] J. K. Iliffe. *Basic Machine Principles*. American Elsevier, Inc., New York, 1968.

[50] J. K. Iliffe. Elements of BLM. *The Computer Journal*, 12(3):251–258, 08 1969.

[51] J. K. Iliffe and Jane G. Jodeit. A Dynamic Storage Allocation Scheme. *The Computer Journal*, 5(3):200–209, 11 1962.

[52] Intel Corporation. *Intel 64 and I -32 rchitectures 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.

[53] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block Oriented Programming: Automating Data-Only Attacks. pages 1868–1882. ACM, 2018.

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

[55] Jane G. Jodeit. Storage organization in programming systems. *Commun. CM*, 11(11):741–746, nov 1968.

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

[57] Antti Kantee. *Flexible operating system internals: the design and implementation of the anykernel and rump kernels*. PhD thesis, 2012.

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

[59] R B; Howarth D J Kilburn, T; Payne. The Atlas Supervisor. 1962.

[60] 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 *CM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 207–220, 2009.

[61] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 437–452, New York, NY, USA, 2017. Association for Computing Machinery.

[62] 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.

[63] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, and Andre DeHon. Low-Fat Pointers: Compact Encoding and Efficient Gate-Level Implementation of Fat Pointers for Spatial Safety and Capability-based Security. In *Proceedings of the 2013 CM SIGS C Conference on Computer & Communications Security*, CCS '13, page 721–732, New York, NY, USA, 2013. Association for Computing Machinery.

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

[65] 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 CM SIGS C Conference on Computer and Communications Security*, pages 1901–1915, 2022.

[66] Jochen Liedtke. Improving IPC by Kernel Design. *SIGOPS Oper. Syst. Rev.*, 27(5):175–188, dec 1993.

[67] 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.

[68] J. H. Saltzer M. D. Schroeder, D. D. Clark and D. H. Wells. Final report of the multics kernel design project. 1977.

[69] Steffen Maass, Mohan Kumar Kumar, Taesoo Kim, Tushar Krishna, and Abhishek Bhattacharjee. ECOTLB: Eventually Consistent TLBs. *CM Trans. rchit. Code Optim.*, 17(4), sep 2020.

[70] 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.

[71] 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.

[72] 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 CM Symposium on Operating Systems Principles (SOSP '11)*, page 115–128, 2011.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

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

[78] 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.

[79] Shravan Narayan, Tal Garfinkel, Evan Johnson, David Thien, Joey Rudek, Michael LeMay, Anjo Vahldiek-Oberwagner, Dean Tullsen, , and Deian Stefan. Segue and ColorGuard: Optimizing SFI Performance and Scalability on Modern x86. In *Proceedings of Workshop on Programming Languages and nalysis for Security (PL S)*, 2022.

[80] Shravan Narayan, Tal Garfinkel, Mohammadkazem Taram, Joey Rudek, Daniel Moghimi, Evan Johnson, Chris Fallin, Anjo Vahldiek-Oberwagner, Michael LeMay, Ravi Sahita, Dean Tullsen, and Deian Stefan. Going beyond the Limits of SFI: Flexible and Secure Hardware-Assisted In-Process Isolation with HFI. In *Proceedings of the 28th CM International Conference on rchitectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 266–281. Association for Computing Machinery, 2023.

[81] 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 nnual Technical Conference (USENIX TC 19)*, pages

269–284, Renton, WA, July 2019. USENIX Association.

[82] 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.

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

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

[85] Ruslan Nikolaev and Godmar Back. VirtuOS: An Operating System with Kernel Virtualization. In *Proceedings of the Twenty-Fourth CM Symposium on Operating Systems Principles*, SOSP '13, pages 116–132. ACM, 2013.

[86] University of Cambridge. The Arm Morello Board. https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri morello.html.

[87] Elliott I. Organick. *Computer System Organization: The B5700/B6700 Series*. Academic Press, 1973.

[88] 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.

[89] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX nnual Technical Conference (USENIX TC 19)*, pages 241–254, July 2019.

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

[91] Mathias Payer and Thomas R. Gross. Fine-Grained User-Space Security Through Virtualization. In *Proceedings of the 7th CM SIGPL N/SIGOPS international conference on Virtual execution environments (VEE)*, pages 157–168, 2011.

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

[93] 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.

[94] 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.

[95] 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.

[96] 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.

[97] 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.

[98] 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.

[99] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. HDFI: Hardware-Assisted Data-Flow Isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17, 2016.

[100] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for Security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295, 2019.

[101] 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.

[102] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proceedings of the 17th nnual International Conference on Supercomputing*, ICS '03, page 160–171, New York, NY, USA, 2003. Association for Computing Machinery.

[103] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on rchitectural Support for Programming Languages and Operating Systems*, ASPLOS XI, page 85–96, New York, NY, USA, 2004. Association for Computing Machinery.

[104] Gregory T. Sullivan, André DeHon, Steven Milburn, Eli Boling, Marco Ciaffi, Jothy Rosenberg, and Andrew Sutherland. The Dover Inherently Secure Processor. In *2017 IEEE International Symposium on Technologies for Homeland Security (HST)*, pages 1–5, 2017.

[105] 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.

[106] 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 CM SIGOPS European workshop*, pages 102–107, 2002.

[107] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013.

[108] Patricia Teller, R. Kenner, and Marc Snir. TLB Consistency on Highly-Parallel Shared-Memory Multiprocessors. pages 184 – 193, 02 1988.

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

[110] Suramya Tomar. Converting video formats with ffmpeg. *Linux Journal*, 2006(146):10, 2006.

[111] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Proceedings of the 37th nnual IEEE/ CM International Symposium on Microarchitecture*, MICRO 37, page 243–254, USA, 2004. IEEE Computer Society.

[112] 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.

[113] 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 CM/IEEE 41st International Symposium on Computer rchitecture (ISC)*, pages 469–480, June 2014.

[114] 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 rchitectures and Compilation Techniques*, PACT '11, page 340–349, USA, 2011. IEEE Computer Society.

[115] 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.

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

[117] 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.

[118] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian Memory Protection. In *Proceedings of the 10th International Conference on rchitectural Support for Programming Languages and Operating Systems*, ASPLOS X, page 304–316, New York, NY, USA, 2002. Association for Computing Machinery.

[119] 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 CM/IEEE 41st International Symposium on Computer rchitecture (ISC)*, pages 457–468, 2014.

[120] 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 nnual IEEE/ CM International Symposium on Microarchitecture*, pages 545–557, 2019.

[121] Zi Yan, Ján Veselý, Guilherme Cox, and Abhishek Bhattacharjee. Hardware Translation Coherence for Virtualized Systems. *SIG RCH Comput. rchit. News*, 45(2):430–443, jun 2017.

[122] 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.

[123] Zachary Yedidia. Lightweight Fault Isolation: Practical, Efficient, and Secure Software Sandboxing. In *Proceedings of the 29th CM International Conference on rchitectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 649–665, New York, NY, USA, 2024. Association for Computing Machinery.

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

[125] 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.

[126] 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.

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