CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization

Robert N. M. Watson*, Jonathan Woodruff*, Peter G. Neumann†, Simon W. Moore*, Jonathan Anderson‡, David Chisnall*, Nirav Dave†, Brooks Davis†, Khilan Gudka*, Ben Laurie§, Steven J. Murdoch¶, Robert Norton*, Michael Roe*, Stacey Son*, Munraj Vadera*
*University of Cambridge, †SRI International, ‡Memorial University, §Google UK Ltd., ¶University College London

***Abstract*—CHERI extends a conventional RISC Instruction-Set Architecture, compiler, and operating system to support fine-grained, capability-based memory protection to mitigate memory-related vulnerabilities in C-language TCBs. We describe how CHERI capabilities can also underpin a hardware-software object-capability model for application compartmentalization that can mitigate broader classes of attack. Prototyped as an extension to the open-source 64-bit BERI RISC FPGA soft-core processor, FreeBSD operating system, and LLVM compiler, we demonstrate multiple orders-of-magnitude improvement in scalability, simplified programmability, and resulting tangible security benefits as compared to compartmentalization based on pure Memory-Management Unit (MMU) designs. We evaluate incrementally deployable CHERI-based compartmentalization using several real-world UNIX libraries and applications.**

I. Introduction

Vulnerability mitigation is a key tenet of contemporary computer-system design. Deployed systems commonly employ two approaches: *exploit mitigation* (which targets attack-vector characteristics such as remote code injection [46]), and *software compartmentalization* (which limits privileges and further attack surfaces available to attackers [25], [41], [27], [53]). Exploit mitigation relies on knowledge of specific attack vectors and avoids application-level source-code modification; for example, stack canaries transparently detect attempts to overwrite return addresses. However, these techniques are often probabilistic and subject to an arms race as attack and defense co-evolve. In contrast, compartmentalization requires structural changes to programs: applications are decomposed into isolated components that are granted selected access to system and application resources, limiting the rights leaked to attackers. Unlike exploit mitigation, compartmentalization can provide protection against yet unknown exploit techniques. The two approaches are complementary and often used together – e.g., OpenSSH [41] and Chromium [42] are frequently compiled with both stack protection and sandboxing.

Unfortunately, these techniques must be retrofitted onto consensus hardware and software models that deemphasize security. This imposes detrimental performance, additional complexity, and programmability problems as stronger protection is layered over weaker substrates. This tendency is clearest at the lowest levels of the stack: widely used CPUs provide little support for fine-grained memory protection, and exhibit poor compartmentalization scalability. As a result, countless research papers explore ways to reintroduce omitted protection features through program transformation. There is little recent work in the area of hardware-software approaches, despite a pressing need for vulnerability mitigation in C-language Trusted Computing Bases (TCBs) such as language runtimes and web browsers, which are neither easily proven correct nor easily replaced with type-safe alternatives.

In prior papers, we have described Capability Hardware Enhanced RISC Instructions (CHERI), a set of incrementally adoptable architectural extensions for scalable, in-address-space memory protection that mitigates exploits via a *hybrid capability-system model* [54], [57], [15]. CHERI supplements the conventional Memory Management Unit (MMU) supporting virtual-memory-based processes with a *capability coprocessor* to implement fine-grained, compiler-directed memory protection. In this paper, we describe how CHERI can also act as the foundation for an *object-capability model* able to support orders of magnitude greater compartmentalization performance, and hence granularity, than current designs.

As with MMU-based systems, CHERI can enforce strong isolation and controlled memory sharing, two prerequisites for compartmentalization, albeit with markedly different scalability and programming properties. We use capabilities to build a *hardware-software domain-transition mechanism* and *programming model* suitable for safe communication between mutually distrusting software. We extend our CHERI ISA and processor prototype with *sealed capabilities* and *hardware-accelerated object invocation*, and extend the CHERI software stack (LLVM compiler [30] and FreeBSD OS [33]) with a *domain-transition calling convention* and a *userspace object-capability model*. While our approach learns from prior capability systems, such as HYDRA [58] and the M-Machine [13], our focus is on *hybridization*: how to incrementally deploy CHERI within current C-language TCBs with source-code and binary compatibility. We have targeted the most security-critical TCBs (e.g., privileged software) and also the most vulnerable (e.g., compression libraries) while avoiding disruption to the remainder of the software stack. In this paper, we:

- Describe a novel hardware-software capability-system architecture supporting incrementally adoptable, fine-grained compartmentalization for C-language TCBs.
- Explore the architecture's practical implications through a fully functional hardware-software prototype based on a 64-bit FPGA soft-core RISC processor, compiler, OS, and example applications. (To facilitate reproducibility, we have open-sourced our hardware and software.)
- Demonstrate an effective and incrementally adoptable hybrid MMU-capability model for compartmentalization, clean composition with OS features such as virtual memory, C-language object capabilities, library compartmentalization, and an orders-of-magnitude performance gain.
- Evaluate the security, complexity, programmability, and performance impacts of the approach, paying particular attention to compatibility concerns that have not been the subject of prior capability-system research.

Throughout, we consider tradeoffs in the hardware-software design space, and their impact on software structure.

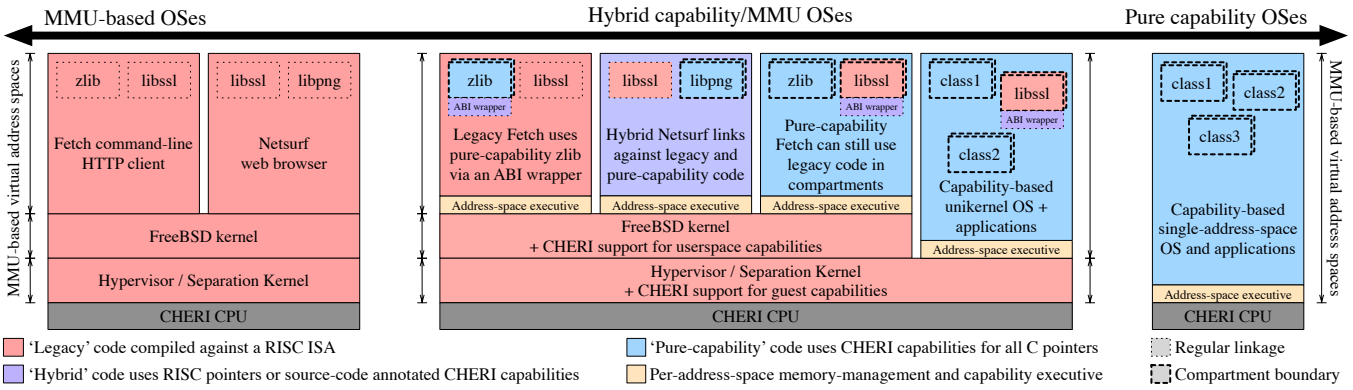

Fig. 1. While the CHERI ISA can support a spectrum of hardware-software architectures, from conventional MMU-based virtualization and OS process models to single address-space capability systems, we focus on hybridization opportunities that allow elements of both approaches to be combined.

II. Approach

The CHERI hardware-software architecture enhances vulnerability mitigation through two capability-based techniques aimed at user-level C-language TCBs:

- **Memory capabilities**, described in prior papers, are implemented by the ISA [57] and compiler [15], providing an incrementally deployable replacement for pointers within address spaces, mitigating memory-based exploits.
- **Object capabilities**, the focus of this paper, are implemented by the operating system over the memory-capability foundation, providing scalable, and likewise incrementally adoptable, software compartmentalization.

Capability systems are hardware, software, or distributed systems designed to implement the *principle of least privilege* [17], [43]. *Capabilities* are unforgeable tokens of authority granting rights to objects in the system; they can be selectively delegated between constrained programs to enforce security policies. While pure capability systems allow access to objects only via capabilities, CHERI is a *hybrid capability system* that relaxes this restriction, providing greater compatibility with existing programs that rely on the assumption of *ambient authority*: the ability to access arbitrary system objects [53]. CHERI also learns from *object-capability systems* (e.g., [40]) that blend object-oriented OS or programming-language facilities with capabilities to protect application-defined objects. Encapsulation and interposition then allow programmers to express a range of security policies [58], [35].

Unlike many historic "pure" hardware capability systems [31], CHERI's hybrid capability-system architecture retains a conventional MMU. This allows a broad range of software models to be implemented, as illustrated in Figure 1. This includes virtual-memory-based OSes such as UNIX, single-address-space capability systems, and most interestingly, hybridized systems that combine elements of both approaches. While CHERI-supported techniques would work equally well within an OS kernel (e.g., to implement microkernels [2]), we choose to focus on *application compartmentalization* due to the large number of lines of code and the key role that user-level TCBs play in attacker entry into systems. Prior work on compartmentalization has assumed that software authors are the well-meaning victims of poor C-language safety, leading to endemic vulnerability in the presence of malicious data – and, in effect, injection of arbitrary malicious code at runtime.

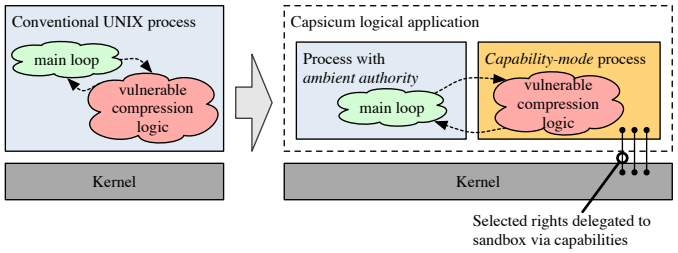

Fig. 2. *Software compartmentalization* decomposes applications into isolated components, limiting rights leaked to successful attackers.

We accept this adversary model, but observe that modest extensions to the CheriBSD class loader would also allow it to tolerate a malicious software supply chain. We begin by briefly introducing capability-based compartmentalization and the CHERI ISA.

Capability-system concepts have proved useful in implementing *software compartmentalization* (a.k.a. *privilege separation*), the mitigation of vulnerabilities by decomposing applications into isolated components – each granted only the rights it requires to operate [25], [41], [27]. Figure 2 illustrates how OS-based compartmentalization can mitigate vulnerabilities: by executing `gzip` compression in a sandbox delegated only capabilities for files being read from and written to, a successful remote-code exploit for a `zlib` vulnerability gains only limited rights to the system as a whole. *Compartmentalization granularity* describes the degree of program decomposition. Fine-grained compartmentalization improves mitigation by virtue of the principle of least privilege: attackers must exploit more vulnerabilities to accomplish rights in the target system.

Compartmentalization at the application layer requires a combination of properties encompassing separation of functionality *between* different applications and certain properties *within* each application (such as modular abstraction with encapsulation and information hiding, separation of privileges, and least privilege), in addition to lower-layer hardware and OS integrity properties that ensure noncompromise of the applications. The overall goal of compartmentalization is to effectively limit functionality and delimit attack surfaces available to attackers, even after seemingly successful exploits. CHERI's improvements to programmability and performance

facilitate not only easier deployment of compartmentalization, but also greater practical granularity, improving resilience against attackers.

Capability-based software compartmentalization is typically implemented via two substrates: OS-based systems such as HYDRA [58], EROS [45], SeL4 [28], and Capsicum [53]; and language-based systems such as E [35], Joe-E [34], and Caja [36]. In both, the underlying substrate provides unforgeable capabilities (file descriptors, communication ports, or language object references), and constraints to prevent bypass of the capability mechanism (e.g., the virtual-memory process model combined with Capsicum's *capability mode* or Joe-E's statically checkable language subset). Finally, some form of escalation mechanism is required – object-capability invocation – to allow protected subsystems to interact while holding distinct sets of rights (e.g., IPC or language-level object encapsulation). CHERI learns from these approaches, adopting ideas about hybridization and OS integration from Capsicum, and ideas about the role of the execution substrate and object orientation from language-based techniques.

The CHERI ISA follows the theme of hybridization to enable incremental adoption by extending a conventional RISC ISA with a capability model to support fine-grained memory protection [54]. CHERI subscribes to the RISC philosophy: instructions are primitives for the compiler rather than the programmer and microcode is eschewed in favor of OS exception handlers. Described in detail in Section IV, CHERI adds *capability registers* with instructions for safe manipulation and use as pointers. Capability integrity is protected even in memory using *tags*. CHERI capability registers describe all regions of the virtual address space accessible to the current thread in much the same way that conventional general-purpose registers contain a working set of pointers. Indeed, CHERI capabilities are designed to represent C-language pointers [15], adopting ideas from the fat-pointer literature to provide adequate expressiveness [23], [39], [18], [29]. With virtual memory delegated as capabilities, each user process on a CHERI system can be considered its own *virtual capability machine*.

In prior work, we have described how CHERI can support strong but incrementally deployable memory protection with the C programming language [57], [15]. This paper builds on that approach by describing how CHERI memory protection can also be used as the foundation for an object-capability model for use in fine-grained software compartmentalization, protecting application-layer constructs and able to express mature policies concerning control and information flow. The clean separation of policy and mechanism in object-capability systems aligns elegantly with the RISC philosophy: with fine-grained protection "fast paths" implemented in hardware, policy definition can be left to the OS, compiler, and application. The resulting hardware-software security model can efficiently implement diverse security policies including hierarchical models (such as sandboxing) and non-hierarchical models (such as communicating but mutually distrusting components).

Evaluating fresh hardware-software approaches is challenging due to the difficulty in establishing baselines and implementation cost for prototypes. To enable a natural baseline, the CHERI processor prototype extends BERI, our FPGA implementation of the classic 64-bit MIPS ISA [20], which runs a range of off-the-shelf, open-source software. We have extended the LLVM compiler [30], FreeBSD OS [33], and several applications to support fine-grained compartmentalization. This allows us to perform side-by-side comparisons, demonstrating performance, security, complexity, and programmability improvements over conventional designs.

The CHERI approach is not without limitations. An ISA-level approach that promotes finer-grained compartmentalization and greater intercommunication is dependent on the hardware substrate to provide strong isolation; however, there is a copious literature on processor side channels (e.g., via shared caches [50]). Our approach also places further dependence on C-language TCBs, including on the compiler to implement protection choices that directly affect security. Past vulnerabilities in language TCBs have been substantial (e.g., vulnerabilities in Java Virtual Machines), and must be considered for our model as well: errors in compilers, low-level memory allocators, parsing of object files, garbage collectors, and so on, are pertinent. We have attempted to mitigate these concerns through formal modeling of the ISA and software TCBs, and through minimizing the software footprint required to implement isolation.

III. The CHERI System Architecture

Our primary goal with CHERI is to extend MMU-based designs with primitives suitable for architecturally clean, expressive, and scalable application compartmentalization for C-language TCBs. Target applications include data-processing libraries, system services, command-line tools, programming-language runtimes, and complex TCB-like applications such as web browsers. To be most effective, CHERI must provide:

Unified protection *that is able to serve complex software stacks such as web browsers that incorporate many libraries and components. This enables clean and safe system composition, and facilitates reasoning about total-system security.*

MMU-based designs have fixed numbers of rings and support a multi-process compartmentalization model that has proven programmer-unfriendly. CHERI supports efficient, synchronous domain switching modeled on function invocation rather than asynchronous inter-process message passing. This enables the obvious compartmentalization strategy to "cut" applications at function-call boundaries (e.g., library APIs).

C-compatible protection *that offers a convenient mapping from C-language constructs (e.g., pointers and structures) and requires as few changes as possible to TCB source code.*

MMU-based systems implement isolation through multiple virtual address spaces, which complicates the programming model and provides only page-granularity protection. C-language constructs such as the stack and data structures are rarely integer multiples of page size or page aligned, but are frequently the linchpin for exploit techniques. CHERI provides fine-grained (byte-granularity) memory protection suitable for use by the compiler in securing these structures, and allows compartmentalization within a single address space.

Scalable protection *that supports large numbers of compartments with continuous interaction and data sharing.*

As numbers of compartments and domain transitions grow, MMU-based techniques scale poorly due to limited Translation Lookaside Buffer (TLB) resources, TLB aliasing from sharing, page granularity, and IPC overhead. CHERI deconflates virtualization from protection, allowing protection granularity to

scale with reduced TLB impact. CHERI optimizes delegation and memory sharing – especially important when targeting latency-sensitive, high-volume intra-application interfaces.

The overall CHERI hardware-software architecture consists of ISA extensions with their CPU implementation, compiler and OS support for both fine-grained memory protection and object capabilities, and applications that utilize memory protection and compartmentalization. Collectively, these changes improve application resilience under a broad range of known (and, thanks to compartmentalization, as-yet undiscovered) vulnerabilities and associated exploit techniques.

A. Instruction-Set Architecture

We have added several features to the CHERI ISA to support compartmentalization. Each capability now includes a *sealed bit* that constrains manipulation of other fields, and a 24-bit *object type* that allows code and data capabilities to be atomically linked. We have also introduced a 2-bit information flow-control model to assist with temporal safety.

B. Operating-System Kernel

Many OS designs can be mapped onto CHERI's hybrid MMU-capability model. At one extreme, an unmodified FreeBSD/MIPS boots without enabling the capability coprocessor. At the other, a clean-slate single-address-space OS might use capabilities for all protection and domain management. By hybridizing these approaches, an OS can utilize the MMU for coarse-grained inter-process separation, and the capability model for fine-grained, intra-process protection and compartmentalization. Our FreeBSD hybridization extensions:

- Initialize the capability coprocessor on boot.
- Maintain tags in virtual-memory operation.
- Delegate suitable initial capabilities to user threads.
- Preserve capability registers when context switching.
- Handle new protection and security exceptions.
- Implement an object-capability model.
- Prevent improper flow of capabilities between processes.
- Offer system interfaces that accept capability arguments.
- Support debugging of capabilities/tagged memory.

C. Compiler

For CHERI memory protection, the compiler generates code that captures object bounds, pointer-integrity properties, and control flow. For CHERI compartmentalization, the compiler supports a new domain-crossing calling convention. In normal operation, modulo compiler bugs, unused values in registers do little harm; with domain crossing, leaked registers not only leak data, but also capabilities. The new calling convention ensures that unused argument and return-value registers, known only to the compiler, are cleared.

D. Object-Capability Model

The heart of CHERI compartmentalization is the object-capability model, supported by the ISA and compiler-directed memory protection, and implemented by the kernel and userspace runtime. As with prior object-capability systems, object encapsulation is the model for isolation, and object invocation provides controlled communication. Capability-based memory protection implements encapsulation, with "sealed" references allowing objects to be referred to and invoked without granting access to private state, and the kernel implements object invocation via hardware-accelerated domain transition.

The CHERI ISA encodes a specific memory-protection model but can support a broad range of hardware-software security models. This is important because a variety of object-capability semantics have been proposed, and we would like to be able to explore many of them on a single platform. For example, prior work has seen disagreement on synchronicity for object-capability invocation: asynchronous primitives allow callers to avoid placing trust in callee termination, but current software designs incorporate strong assumptions of synchronicity [45]. Software could implement either model on CHERI; however, we choose to provide a simple, synchronous mechanism modeled on function calls to ease inserting protection-domain boundaries into existing call graphs:

- *Object-capability invocation* pushes execution state from the caller object onto a trusted stack, unseals the argument object, and performs a secure domain transition to it.
- *Object-capability return* pops the caller from the trusted stack and performs a secure domain transition back to it.

Safe transition between security domains is the joint responsibility of several "parties": the ISA provides underlying memory protection that ensures isolation, both between compartments and from the TCB; the kernel's handlers implement domain transition that supports both asymmetric and mutual distrust; the compiler and application ensure that protected state is maintained, and that only intended data and capabilities are passed via arguments or return values.

Within each process, a userspace *address-space executive*, with code spanning `libc` and `libcheri`, is responsible for security-critical TCB functions such as memory management and class loading. The executive configures memory protection to implement isolation, safely allocates (and reallocates; e.g., via garbage collection) memory and objects, loads class code, and passes initial capabilities for both memory and communications into new objects. Useful comparison can be made between the address-space executive and both microkernels and language security-model runtimes (e.g., Java). Unlike a microkernel, the executive resides within a UNIX process; like Java support for native code, the executive is responsible for coordinating communication between compartments and general OS services. Unlike the Java security model, code injection attacks are part of the threat model, and containment is maintained even if unexpected instructions enter execution.

IV. Implementation

To explore and evaluate the CHERI approach, we have implemented a complete hardware-software prototype based on off-the-shelf, open-source designs:

- The **CHERI ISA** and **CHERI processor prototype** provide hardware-accelerated capability primitives able to support efficient software compartmentalization.
- The **CheriBSD kernel** implements capability-based intra-process memory protection and domain transition.
- The **CHERI Clang/LLVM** compiler supports a new capability ABI and object-capability calling convention.
- **CheriBSD's userspace** includes a compartmentalization library, and classes that provide services to compartments.

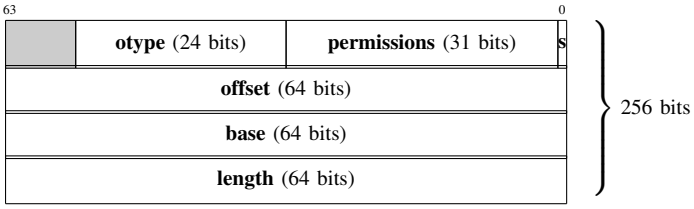

Fig. 3. ISA-level representation of a 256-bit CHERI capability

TABLE I. CAPABILITY INSTRUCTIONS

Instruction	Description	Priv.	Soft.
CGetBase	Get capability base		
CGetOffset	Get capability offset		
CGetLen	Get capability length		
CGetTag	Get capability tag		
CGetPerm	Get capability permissions		
CToPtr	Convert capability to pointer		
CPtrCmp	Compare two capabilities		
CIncBase	Increment capability base		
CSetLen	Set capability length		
CClearTag	Clear capability tag		
CSetOffset	Set capability offset		
CFromPtr	Convert pointer to capability		
CSC	Store capability via capability		
CLC	Load capability via capability		
CL[BHWD][U]	Load data via capability		
CS[BHWD]	Store data via capability		
CLL[WD]	Load linked data via capability		
CSC[WD]	Store conditional data via capability		
CGetPCC	Get program-counter capability		
CBTU	Branch if capability tag unset		
CBTS	Branch if capability tag set		
CJR	Capability jump		
CJALR	Capability jump and link		
CGetCause	Get capability cause register	P	
CSetCause	Set capability cause register	P	
CGetSealed	**Get capability sealed bit**		
CGetType	**Get capability type**		
CSeal	**Seal capability**		
CUnseal	**Unseal capability**		
CCheckPerm	**Check capability permissions**		
CCheckType	**Check capability type**		
CCall	**Invoke object capability**		S
CReturn	**Return from object capability**		S

P: Privileged instruction available only to the supervisor.
S: Implemented in part or fully via an exception to the supervisor.

- Several **UNIX libraries and applications** utilize fine-grained, object-capability-based compartmentalization.

A. Instruction-Set Architecture (ISA)

CHERI enhances the 64-bit MIPS ISA with compiler-managed, capability-based, intra-address-space memory protection[1]. With only modest extensions, it can also support an efficient, software-defined object-capability model. We briefly review the CHERI ISA before describing these extensions.

CHERI defines a set of *capability registers* similar in structure to fat pointers (see Figure 3). The capability register file is accessed using *capability instructions* (see Table I), which allow capabilities to be loaded and stored from memory, to be inspected and manipulated (e.g., to get or set the length), to be dereferenced via *load* and *store* instructions, and to be the target of *jump* and *branch* instructions. Access via a capability is subject to a validity check on its *tag*, relocation relative to its *base* and *offset*, bounds checking relative to its base and *length*, and permission checking. *Capability permissions* control what operations can be performed via a capability (see Table II). Most registers are available to compiler and OS-defined Application Binary Interfaces (ABIs), but certain registers are reserved in the ISA (see Table III). The *program-counter capability* (`$pcc`) extends the MIPS *program counter* (`$pc`) to constrain code execution. The *default data capability* (`$ddc`) interposes on conventional MIPS loads and stores; a suitable `$ddc` can entirely disallow MIPS-ISA memory access.

[1] CHERI is prototyped as an extension to the 64-bit MIPS ISA, but its concepts should apply, with localization, to any RISC ISA (e.g., ARMv8 or RISC-V). Many surface design choices mirror MIPS (e.g., 32 registers and a software-only stack), and would likely be made differently for other ISAs.

TABLE II. CAPABILITY PERMISSIONS

Permission	Description
Permit_Execute	Fetch instructions
Permit_Load	Load data
Permit_Store	Store data
Permit_Load_Capability	Load capability
Permit_Store_Capability	Store capability
Permit_Exception	Access to exception registers
Global	**Capability has global scope**
Permit_Store_Local	**Can store non-global capabilities**
Permit_Seal	**Can be used to seal objects**

TABLE III. CAPABILITY REGISTERS

Register	Description	Priv.	ISA	ABI
`$pcc`	Program-counter capability		I	
`$ddc`	MIPS default data capability		I	
`$stc`	Stack capability			A
`$c3–$c10`	Argument, return capabilities			A
`$c11–$c16`	Caller-save registers			A
`$c17–$c24`	Callee-save registers			A
`$kr1c`	Exception-handling capability	P	I	
`$kr2c`	Exception-handling capability	P	I	
`$kcc`	Kernel code capability	P	I	
`$kdc`	Kernel data capability	P	I	
`$epcc`	Exception program-counter capability	P	I	
`$scc`	**Sealed code capability**			A
`$sdc`	**Sealed data capability**			A
`$idc`	**Invoked data capability**			A

P: Privileged register available only to the supervisor. I: Defined by the Instruction-Set Architecture (ISA). A: Defined by the Application Binary Interface (ABI).

CHERI capabilities are unforgeable by virtue of *guarded manipulation* and *tagged memory*. Guarded manipulation ensures that instructions permit only monotonic non-increase in rights – i.e., with respect to the memory region described, permissions granted, and so on. Tagged memory associates a 1-bit tag with each physical memory location that can hold a capability, indicating the presence of a valid capability. Stores to, and loads from, capabilities in memory are atomic with their tags, allowing safe concurrent access from multiple cores. The set of memory locations accessible to executing code is the transitive closure of capabilities in its capability register file, and any further capabilities reachable through those capabilities. Any capabilities held during userspace execution are descended from those granted to the supervisor at boot, and later from the supervisor to userspace.

Compiler-directed, fine-grained, capability-oriented memory protection within a virtual address space can serve as a natural isolation mechanism within user processes, and hence a foundation for compartmentalization. An object-capability model could also be constructed using CHERI's hybrid features without any ISA extension: user threads with access to

(perhaps overlapping) subsets of the user address space could invoke the software supervisor, which holds a superset of their rights, via system calls to implement asymmetric or mutual distrust. We choose to extend the ISA for several reasons:

- To treat object capabilities as first-class citizens in C as we do memory capabilities – for example, by permitting object-capability references to replace function pointers.
- To keep important programmer- and compiler-defined paths in userspace – for example, avoiding system calls for additional permission or type checks.
- To avoid the kernel needing to maintain parallel structures (e.g., object registries) to implement encapsulation.
- To avoid the need to expose conventional kernel system calls to userspace compartments; while sometimes useful, this is antithetical to kernel attack-surface reduction.
- To allow limits on capability propagation to reduce the cost of (and need for) garbage collection, and to avoid temporal safety issues.

We therefore implement extensions to CHERI memory protection: sealed capabilities with object types, instructions for capability invocation, instructions for efficient permission and type checking, and new permissions enforcing a simple information flow-control policy to limit capability propagation.

1) Object Capabilities: Whereas CHERI memory capabilities refer to bounded regions of memory within the virtual address space, *object capabilities* refer to software-defined objects whose invocation will trigger a protection-domain switch. The object-capability mechanism provides *encapsulation*, which restricts not just caller access to callee-private data, but also callee access to caller-private data, providing a safe foundation for mutual distrust. CHERI object capabilities are invoked in pairs: a *sealed code capability* describes the code to be executed when an object is invoked (i.e., the *class*), and a *sealed data capability* describes its instance-specific data.

To prevent callers from manipulating the internal state of object capabilities (which would violate encapsulation), an object's code and data capabilities are both *sealed*, indicated by a new **sealed** bit in the capability. Sealed code and data capabilities are differentiated by whether or not the `Permit_Execute` permission is set. Sealed capabilities are entirely immutable: any attempt to manipulate a field of a sealed capability will throw an exception. The sealed bit also prohibits capability dereference: sealed capabilities may not be used to load, store, or execute instructions, providing encapsulation.

Sealed code and data capabilities are atomically linked by a new 24-bit capability field, **otype**, which contains a software-defined *object type* that must be identical for a pair of code and data capabilities to be accepted for invocation. Capabilities are sealed using the new `CSeal` instruction, which accepts two capability-register arguments: the code or data memory capability to be sealed, and a second capability with the `Permit_Seal` permission set. The effective virtual address of a capability with `Permit_Seal` set is treated as a type (provided that it is smaller than 24-bits). Although this arrangement conflates the type space and address space, we expect that software implementations will divorce the capability type space from the memory space through use of permissions.

2) Object-Capability Invocation: Object-capability invocation is implemented via two new instructions: `CCall`, which invokes a sealed code/data-capability pair, and `CReturn`, which returns to the invoking context. In order to support a wide variety of software behaviors, the CHERI ISA relies on software exception handlers to partially implement both instructions, allowing the supervisor to implement both synchronous ("call-return") and asynchronous ("message passing") semantics.

To exploit hardware parallelism, the CHERI ISA allows certain checks (for sealing, suitable permissions, and matching types) to be performed by `CCall`, with the exception vector and exception code selected based on their results. `CReturn` simply triggers a software exception handler without checks, and may be eschewed entirely in asynchronous implementations where `CCall` is effectively a message-send primitive. The `CCall` and `CReturn` mechanisms described by the ISA are not sufficient, in isolation, to implement secure protection-domain transition: the software runtime (including the supervisor, userspace runtime, and compiled code within objects) must ensure that memory allocation and capability distribution implement any required isolation, and that both the general-purpose and capability register files have been flushed of private data and rights prior to invocation or after return.

`CUnseal`, another new instruction, allows authorized software to remove the **sealed** bit if it also holds a capability usable to seal the type. This "escape valve" is used by the `CCall` exception handler to unseal the sealed code and data capabilities passed by arguments. It can also be used by a userspace class to unseal argument objects that are not automatically unsealed by the invocation mechanism. The CHERI ISA itself will never automatically unseal capabilities, avoiding potential risks associated with unintended *amplification* (e.g., as could occur in [58]). Two assertion instructions are introduced to allow the userspace runtime to efficiently determine that argument capabilities have desired permissions (`CCheckPerm`) or have a suitable object type (`CCheckType`). These can be combined with software-defined permission bits on capabilities to control access to specific methods on the object, and to determine that object capabilities passed as arguments have suitable types.

3) Global vs. Local Capabilities: CHERI spatial protection does not natively prevent use-after-free or other temporal safety violations; these are controlled by program, language, or runtime mechanisms – e.g., software invariants or garbage collection. When executing within a single security domain, rapid memory reuse does not constitute a vulnerability in the model. However, when memory is passed between protection domains, memory reuse could lead to significant temporal vulnerability. This is particularly relevant to the C idiom of passing pointers to on-stack data structures as function arguments.

To assist the software security model in addressing temporal issues, we have extended the CHERI ISA with a 2-bit information flow-control model that marks capabilities as either global or local. *Global capabilities*, identified by the new **Global** permission, may be stored via any writable memory capability. *Local capabilities*, without **Global** set, may be stored only via capabilities that themselves have the new **Permit_Store_Local** permission set. The global/local mechanism restricts only the flow of capabilities, not data.

This primitive limits the propagation of selected capabilities (and their descendants via guarded manipulation) to specified memory. In CheriBSD, stack capabilities (and hence stack-derived capabilities) are local, heap capabilities permit storing only global capabilities, and `CCall` blocks delegation

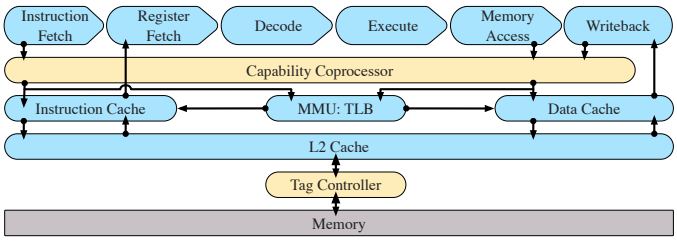

Fig. 4. BERI pipeline with capability coprocessor

TABLE V. CHERIBSD KERNEL CODE CHANGES

Component	Files Modified	Lines Added	Lines Removed
Headers	19	1424	11
CHERI initialization	2	49	4
Context management	2	392	10
Exception handling	3	574	90
Memory copying	2	122	0
Virtual memory	5	398	27
Object capabilities	2	883	0
System calls	2	76	0
Signal delivery	3	327	71
Process monitoring/debugging	3	298	0
Kernel debugger	2	264	0

of local capabilities. This in effect requires that memory arguments to invocation (and return) be heap allocated, exposing delegable memory to global non-reuse, revocation, and garbage-collection policies – and preventing stack memory from being passed by reference. The feature could also be used to build more complex models, such as enforcing bounded delegation of capabilities for the duration of an invocation.

B. CHERI Processor Prototype

The open-source BERI/CHERI FPGA soft-core processor [51] includes a capability coprocessor that implements the CHERI ISA's capability instructions and tagged physical memory (see Figure 4). Only minor additions were required to implement support for software-defined object capabilities:

- `CCall` and `CReturn` instructions trigger a new fast-path exception vector, similar to the TLB-miss exception handler, to enable an optimized protection-domain switch.
- Two capability fields: a 1-bit **sealed** field indicates that a capability is sealed, and a 24-bit **otype** field holds a software-managed object type.
- Hardware-defined permission bits support the local/global information-flow policy and sealed objects.
- Instructions allow sealing, unsealing, permission checking, and type checking.

With the existing capability coprocessor, the costs of these additions were negligible in the hardware design in terms of implementation resources in FPGA and the critical path, and consumed only a small amount of opcode space in the ISA.

C. CheriBSD Kernel

We have extended CheriBSD, an adaptation of FreeBSD that supports CHERI memory protection, to implement a lightweight object-capability model for application compartmentalization. As our focus is on applications rather than microkernel decomposition, we minimized kernel modification,

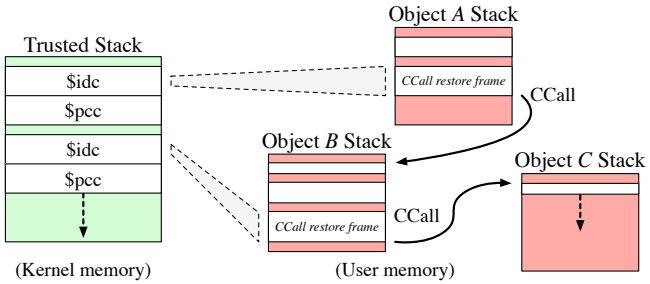

Fig. 5. The trusted stack records a secure return path across object invocation, linking a set of disjoint stacks used in different protection domains.

```
/* ISA validation of CCall arguments. */
if ((!$scc.valid || !$sdc.valid) || !$scc.sealed ||
    !$sdc.sealed) || ($scc.type != $sdc.type) ||
    !($scc.perms & EXECUTE) || ($sdc.perms & EXECUTE) ||
    ($scc.offset >= $scc.length))
        throw_exception();

/* Software exception handler. */
if (capregs.has_local_args())
        throw_exception();
if (trusted_stack.full())
        throw_exception();
trusted_stack.push($epcc);
trusted_stack.push($idc);
$epcc = cunseal($kcc, $scc);
$idc = cunseal($kdc, $sdc);
mipsregs.clear_nonargument();
capregs.clear_nonargument();
```

Fig. 6. Pseudocode for the `CCall` instruction and exception handler

even building the kernel with an out-of-the-box MIPS compiler, and relied on only a small number of lines of CHERI-aware assembly. The CheriBSD kernel initializes the capability coprocessor, sets up and maintains kernel and user capability contexts, implements capability-aware virtual memory, and now also implements object-capability invocation and return. These changes are summarized in Table IV. Statistics on the number of lines of code affected by these changes are summarized in Table V: a negligible impact on the overall kernel of roughly 12.6M lines. Similar changes would likely allow FreeBSD to support other tagged-memory security models, such as those in the CRASH-SAFE design [14].

The CheriBSD object-capability model revolves around the notion of a per-thread *trusted stack* that links a chain of disjoint, per-compartment stacks used by each object executing in the thread, illustrated in Figure 5. The trusted stack is initially empty, with the first thread of the first process executing with ambient authority (global `$pcc`, `$ddc`, and `$stc`). On each invocation, `CCall` will push a new entry onto the trusted stack; on each return, `CReturn` will pop the last entry off. Stack frames consist of two saved capability registers: the caller's program-counter capability, incremented by one instruction to return after `CCall`; and the caller's invoked data capability, which allows callers to preserve state required to restart execution from an otherwise cleared register file. Callers will typically point `$idc` at a small data structure on the caller stack, which will save `$stc`, `$ddc`, and so on.

Figure 6 illustrates pseudocode for `CCall`, which must check that the called code and data capabilities are valid and properly sealed, and have matching types and suitable permissions. It also checks that argument capabilities either

TABLE IV. CHERIBSD KERNEL CHANGES TO SUPPORT USERSPACE CAPABILITIES

Subsystem	Description
Thread contexts	Capability register-file state is maintained for both user and kernel thread contexts. Following `exec()`, the `$ddc`, `$pcc`, and `$stc` registers of the first thread of the process are initialized to grant full access to the user virtual-address space, and the right to perform system calls. Thread creation inherits the capability-register state of the parent thread, as is the case for general-purpose registers. Legacy binaries never manipulate this capability state, and hence execute without modification.
Context switching	As with ordinary registers, capability registers are saved when a user thread enters the kernel, or a kernel context switch occurs.
Exception handling	When an exception fires, the MIPS ISA preserves `$pc` in `$epc`, installing a vector address in its place. Assembly-language handlers use two ABI-reserved registers, `$k0` and `$k1`. The CHERI ISA similarly preserves `$pcc` in `$epcc`; CheriBSD's handlers save the preempted `$ddc`, and install `$kdc` so that otherwise unmodified MIPS handlers can be used. To avoid leaked rights, CHERI's `$kr1c` and `$kr2c` are protected by the ISA, not just the ABI.
CHERI exceptions	Exceptions may be generated when dereferencing an invalid capability, or violating guarded-manipulation rules; these are mapped to a new `SIGPROT` signal, and delivered via the normal UNIX signal mechanism.
Object capabilities	CheriBSD implements an object-capability scheme supporting synchronous invocation via *trusted stacks* that tracks each user thread's invocations. Software portions of invocation and return are implemented via `CCall` and `CReturn` exception handlers.
System calls	The kernel rejects system calls when the executing userspace code capability does not have the software-defined `PERM_SYSCALL` permission. This allows the userspace runtime to limit direct system-call accesses from untrustworthy objects.
Signal delivery	Signal handlers receive capability registers via an extended register frame. Handlers execute in an ambient context on a signal stack rather than on a borrowed stack from untrustworthy objects. Language- or runtime-specific handlers might allow sandboxed code to catch the exception, unwind the trusted stack, or terminate the object or process.

```
/* Software exception handler. */
if (capregs.has_local_retval())
        throw_exception();
if (!trusted_stack.empty()) {
        throw_exception();
$idc = trusted_stack.pop();
$epcc = trusted_stack.pop();
mipsregs.clear_nonreturnval();
capregs.clear_nonreturnval();
```

Fig. 7. Pseudocode for the `CReturn` exception handler

either untagged or have the **Global** permission. It pushes the current `$pcc` and `$idc` onto the trusted stack, and installs unsealed versions of the new code and data capabilities in `$pcc` and `$idc`. `CCall` clears any non-argument general-purpose or capability registers; this could be done by the caller and callee, but clearing in the TCB allows both sides to rely on it always happening, avoiding the need to clear registers in both to prevent leakage or accidental use of leaked data or capabilities. In the event of an error – e.g., a data-code type-check failure or trusted-stack overflow – the handler delivers a UNIX signal.

Figure 7 illustrates pseudocode for `CReturn`, which has the simpler tasks of validating that any returned capability is global or `NULL`, clearing of non-return capabilities, and popping and restoring `$pcc` and `$idc`. Likewise, any errors are handled by a full context switch to the kernel and signal delivery.

One further consideration is the availability of system calls to compartmentalized user code. Many models could be used, including a capability-based system-call ABI in which the kernel enforces userspace memory protection for a capability-safe subset of system calls (e.g., querying the time of day). In the interests of minimalism, CheriBSD offers only a conventional MIPS n64 ABI system-call interface, and accepts system calls only from classes that have the software-defined `User_Syscall` permission. The userspace runtime can thus deny ambient authority associated with the open-ended system-call interface; compartmentalized code must instead request kernel services via system objects that constrain access. This has the further benefit of allowing the userspace runtime to eliminate the system-call interface from the attack surface, if desirable.

D. CHERI Clang/LLVM

Changes to the Clang/LLVM compiler to support CHERI's memory-protection and compartmentalization features are summarized in Table VI. Clang changes include adding language-level support for capabilities and eliminating assumptions that pointers are interchangeable with integers. LLVM changes are split into those specific to the MIPS back end, supporting the CHERI ISA and ABIs, and those updating assumptions that the target-agnostic code generator and mid-level optimizers make about pointers. Due to changes in effective pointer size and register-file use, compiling with capability support necessarily changes the ABI including data-structure layout and calling convention. We define two CHERI-aware ABIs:

- The *hybrid ABI* has a goal of maximum compatibility with the MIPS n64 ABI: only specially annotated pointers are compiled as capabilities. Code compiled against this model can be cleanly mixed with unmodified MIPS code, except where capabilities are explicitly used.
- The *capability ABI* has a goal of maximum protection: all pointers are compiled as capabilities unless explicitly annotated. Interoperability with MIPS code requires ABI wrappers, typically compiled using the hybrid ABI.

The latter ABI is used primarily within objects, whereas CheriBSD code outside of the compartmentalized environment is compiled to the MIPS n64 ABI or hybrid ABI to provide compatibility with existing libraries and the kernel ABI.

By default, the CHERI LLVM compiler generates code to provide precise memory protection: capabilities are used wherever possible to limit accidental buffer overruns, protect pointers (including those used in control flow) from corruption in memory, and so on. However, the underlying assumption is one of mutual trust: callees and callers make no attempt to limit leakage of data or capabilities between them, as they are within the same protection domain. When crossing protection-domain boundaries, substantially more care is required. Leaking a capability from a caller to a callee (or vice versa) could have serious integrity, confidentiality, and availability implications.

TABLE VI. COMPILER CODE CHANGES, EXCLUDING TESTS

Component	Files		Lines		
	Modified	Total	Added	Removed	Total
Clang front end	65	1,343	1,779	99	839,356
LLVM MIPS back end	49	134	3,232	182	53,308
LLVM target-independent	69	2,643	2,428	132	1,244,021
Total	241	3,949	10,463	535	2,136,685

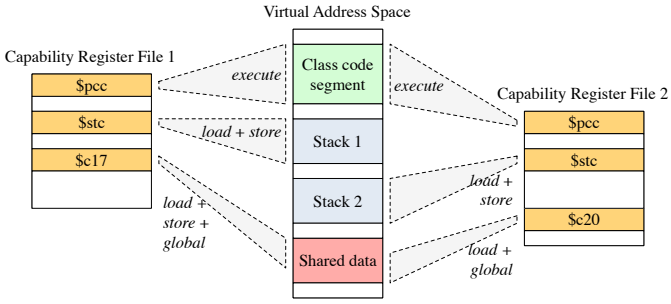

Fig. 8. Capability register files describe the rights of a user thread, and can be used to implement both isolation and controlled memory sharing.

The compiler implements a new calling convention, `CHERI_CCall`, used for annotated functions that can be invoked across domains. Only the compiler, with access to the function type, is aware of which argument and return-value registers are used. It is therefore responsible for generating code that clears unused argument registers in the caller context, and unused return registers in the callee context. `CCall` and `CReturn` are responsible for clearing all other registers.

E. CheriBSD Userspace

We have extended the CheriBSD userspace in several ways:

- `libcheri` loads and run-time links classes, instantiates objects, provides common caller stub code for object invocation, and implements the system classes.
- The *system classes* provide object-capability wrappers for runtime services (e.g., heap allocation and `printf()`), and for delegated OS services such as file descriptors.
- A version of the C start-up code (CSU) provides a "landing pad" for classes, which handles object constructors and callee vtable interpretation during invocation.
- `libc_cheri` is linked into compartmentalized code, and provides caller stubs for system-class methods.

Code outside of the compartmentalized environment is compiled either for the MIPS n64 ABI or the hybrid ABI. Within the compartmentalized environment, the capability ABI is used, compiling all pointers as capabilities. Figure 8 illustrates a possible user address space set up by `libcheri`, which has loaded a single class with two active objects, each executing with its own stack, but accessing overlapping shared data. Through appropriate delegation of capabilities initially and at runtime, access to global state and communication between objects is controlled by the memory protection model.

Object-capability invocation occurs via the `cheri_invoke` function, which accepts two capability pointers representing the sealed code and data capabilities, a method number, and capability and data arguments. It bundles the current execution state context for preservation into memory pointed to by `$idc`, and then executes `CCall`, restoring state from `$idc` upon return.

`libcheri` prohibits use of system calls within compartmentalized code, requiring compartments to instead invoke system classes. The `cheri_fd` system class allows file descriptors to be delegated to compartments: when a method is invoked on a `cheri_fd`, `CCall` reinstalls ambient authority for its execution, allowing system calls such as `read` and `write`. The file-descriptor number is embedded in the sealed data capability of the file-descriptor object, preventing the caller from tampering with the descriptor number.

V. APPLICATION CASE STUDIES

CHERI brings two substantial improvements to compartmentalization relative to process-based approaches: (1) programmability improvements stemming from a single address space, tight C-language integration, and an object-capability model; and (2) scalability improvements due to a hardware-software approach that provides fast, low-latency communication, and reduced cache and TLB footprints due to reduced dependence on virtual addressing. Both aspects support more extensive deployment of compartmentalization, which will contribute to more and better application decomposition, and improve vulnerability mitigation through closer approximation of the principle of least privilege. Our application case studies employ (and naturally compose) a variety of compartmentalization design patterns:

Sandboxing employs compartments with very few delegated rights, and is typically used when processing untrustworthy data using less robust code (e.g., in rendering downloaded images), or for isolating untrustworthy code (e.g., downloaded code in a web browser) [25], [19], [35], [48], [42].

Assured pipelines employ a series of compartments linked by communication channels to perform staged processing of data while limiting the access of (and exposure of) compartments in the chain [11]. This technique can be used to link the interfaces of firewalls or guards via steps such as data normalization, malware scanning, and so on.

Horizontal compartmentalization compartmentalizes multiple instances of the same processing performed on different data instances. At its most granular, this could mean reserving sandboxes for particular downloads or remote sites, but more coarse-grained approaches might distribute different security interests over a small number of sandboxes for reasons of cost (e.g., as is done with tabs in the Chromium web browser [42]).

Vertical compartmentalization associates compartments with particular stages in the processing of the same flow of data, taking on a structure similar to an assured pipeline but with fewer constraints. This might be appropriate in, for example, compartmentalized web-page rendering: each `iframe` might be encapsulated in a compartment, but with further nested compartments being created to render nested `iframes`.

Temporal compartmentalization is concerned with the reuse of objects across different consumers or instances of data. We are concerned with both the potential impact of prior-task residue on the integrity and availability of the current task, and with the potential impact on confidentiality of current-task residue available to future tasks. Limiting object reuse mitigates both concerns, but imposes semantic cost due to loss of state continuity, and overhead due to object re-instantiation.

Work-bounded compartmentalization limits an attacker's ability to prevent forward progress by exploiting control flow bugs (e.g., by triggering an infinite loop). By limiting the amount of work a compartment may perform per invocation, denial-of-service attacks can be mitigated.

Library compartmentalization occurs when software libraries utilize compartmentalization internally, regardless of the application model [53]. This approach can improve the security of all applications linked against the library – e.g., sandboxing within `zlib` benefits any application that uses it.

Compartments have varying trust relationships. The sandboxing pattern is premised on *asymmetric distrust*: applications do not trust sandboxed components, but sandboxed components must trust the containing application. *Mutual distrust* is more challenging: two components must interact to accomplish some larger goal, while distrusting any input from the other.

MMU-based techniques implement strong isolation via extensions to the process model (e.g., Capsicum's capability mode, or SELinux-restricted processes), as well as convenient delegation of OS resources, such as files and sockets, to compartments. However, they provide fewer tools when the resources of interest are within the application itself – e.g., for limiting in-application access to an in-memory database, or in preventing an exploited HTTP vulnerability from leaking TLS keying material for another connection. This is because process-based techniques rely on Inter-Process Communication (IPC) for communication between compartments, forcing use of message passing or page-granularity shared memory. Processes and IPC also suffer poor scalability, limiting applications to extremely modest numbers of compartments (perhaps dozens). CHERI complements process-based approaches through stronger support for inter-domain communication.

A key design question is how the programmer will expose compartmentalization choices to the implementation. For MMU-based designs, this is via system calls that request multiple processes, and explicit IPC calls – often implementing an object-capability model via message passing and shared memory. CHERI memory protection benefits from tight language integration: C types and memory allocation provide fine-grained information required to set up CHERI capabilities enforcing language-level goals. As C does not have a native object model, we are unable to exploit this as a natural source of object-capability information – in contrast to object-capability work based on, for example, Java. However, we observe that conflating protection domains and language-level objects also has limitations: encapsulation for the purposes of software engineering will rarely align with application security goals, causing both to lose out. Instead, we require explicit encapsulation of compartmentalized code in loadable classes, with functions annotated for use as object methods. A small amount of code must be written to provide interfaces between the ambient environment and compartmentalized code. This code must be carefully crafted to ensure overall security properties are met. There is a significant literature on writing safe object-capability software, with the thoughts of Miller [35] and Mettler, et al., particularly relevant to CHERI [34].

Our case studies are selected explore the performance, semantics, and relative merits of CHERI in comparison to existing MMU-based techniques as represented by Capsicum. They explore cases where existing compartmentalization has limited scope because of its focus on OS-level primitives, such as file descriptors – and where utilizing IPC-based communication would impose potentially prohibitive performance overheads. Concerns with compartmentalization-boundary placement, ease of adaptation, and effective mitigation are common across all applications we have compartmentalized, and lessons from these case studies have broad applicability. Our key questions are: Does CHERI accomplish its performance goals, providing greater scalability? Does CHERI accomplish its programmability goals, facilitating further compartmentalization? Finally, are there opportunities to hybridize not just the hardware protection models, but also OS-level and application-level compartmentalization models, for greater overall benefit? We believe the answer to all three of these questions to be yes.

A. `zlib`/`gzip`

The UNIX `gzip` compression tool, based on the `zlib` library, provides a simple but powerful case study. `gzip` accepts a series of filename arguments, compressing the contents of each file. As `zlib` has historically suffered serious security vulnerabilities, sandboxing its compression code provides benefit to any applications linked to it. Unfortunately, `zlib`'s APIs prove unconducive: while Capsicum supports safe and efficient delegation of OS resources, such as the file descriptors of files being compressed by `gzip`, it cannot provide efficient support for `zlib`'s memory-buffer APIs. While Capsicum can support library compartmentalization, in this case – as in many cases involving data-processing libraries, such as similar video decompression libraries – API structure would impose unacceptable overheads, linear on data size, due to passing byte streams over IPC rather than simply passing pointer arguments.

CHERI, in contrast, is designed around memory delegation, making it possible to compare application and library compartmentalization within a single framework. We have extended each of `zlib` and `gzip` to utilize both Capsicum or CHERI sandboxing to compare their compatibility and performance properties. The modified `zlib`, although using capabilities and compartmentalization internally, is ABI-compatible with the original. `zlib` is typical of many compression and data-processing libraries, including image and video CODECs: it was written when performance was an overriding goal, but malicious data was rare. Ideally, an affordable compartmentalization technology should scale to isolating the processing of individual images, video frames, and audio samples.

B. *tcpdump*

`tcpdump` is a widely used packet analyzer that sniffs network interfaces and parses packets to provide a human-readable description. It is a classic example of a high-risk network application: `tcpdump` requires OS privilege, and performs C-language parsing of data received from potentially malicious parties. It has experienced many past vulnerabilities due to the large number of hand-crafted packet parsers.

To understand how compartmentalization affects `tcpdump`, we analyzed 29 vulnerabilities from 1999 to 2015 described in MITRE's Common Vulnerabilities and Exposures (CVE) [47] list. With one exception, all vulnerabilities were found in printing functions. Table VII compares mitigation across sets of vulnerabilities with common impacts for uncompartmentalized `tcpdump`, `tcpdump` with Capsicum sandboxing as shipped in FreeBSD, and our CHERI-compartmentalized version.

TABLE VII. SUMMARY OF `tcpdump` CVE VULNERABILITIES AND THEIR MITIGATION VIA COMPARTMENTALIZATION

No. of CVEs	Vulnerability Type	Impact		
		No Sandboxing	Capsicum	CHERI
11[†]	Input validation	Privileged process DoS-loop	Sandbox process DoS-loop	Sandbox object restart
10[††]	Buffer overflow/underflow, Unsafe memory copy, Unsafe snprintf	Privileged process code injection	Sandbox process code injection	Sandbox object restart
6[‡]	Buffer overflow/underflow	Privileged process DoS-crash/info leak	Sandbox process DoS-crash/info leak	Sandbox object restart/info leak
1[‡‡]	Input validation	Privileged process DoS-stack	Sandbox process DoS-stack	Sandbox object restart
1[‡‡‡]	NULL function pointer	Privileged process DoS-crash	Sandbox process DoS-crash	Sandbox object restart

[‡]2014-876{9,8},2004-0{183,184,057,055} [‡‡]2003-1029 [‡‡‡]2015-2155 [†]2005-12{81,80,79,78,67},2003-0{989,145,108,093},2000-0333,1999-1024 [††]Misc: 2015-215{4,3}; Buffer overflow/underflow: 2015-0261,2014-9140,2007-1218,2002-0380,2001-1279,2000-1026; Unsafe snprintf: 2007-3798; Unsafe memory copy: 2002-1350

Previous compartmentalization of `tcpdump` mitigates some of these issues: Capsicum limits access to the input file/packet stream and output file or standard output. This substantially constrains the effects of a successful exploit, from full root access to a small number of privileges. However, those privileges continue to offer significant power to the attacker. An attacker can crash `tcpdump` (marked DoS-crash in the table), render it inoperative by triggering an infinite loop (DoS-loop and DoS-stack), cause it to lose packets by triggering excessively expensive decoding operations, gain access to prior data observed by the tcpdump session that may remain in memory, or even take control of execution, causing further packet data to be obscured or suppressed. Such blinding attacks are commonly employed in capture-the-flag events to disrupt traffic analysis prior to deploying more powerful exploits.

CHERI memory protection gives us automatic mitigation of vulnerabilities resulting from buffer overflows and underflows: most unsafe accesses turn into ISA-level length exceptions, resulting in a signal being delivered to the process. Combined with CHERI compartmentalization, a signal resulting from an unsafe operation within a sandbox can be gracefully handled by the caller of the sandbox, which can then terminate or reset the sandbox and continue parsing packets.

We have compartmentalized `tcpdump`'s packet-dissection code using horizontal, vertical, temporal, and work-bounded compartmentalization patterns. We have implemented horizontal compartmentalization by adding a mode with two trivial packet selectors, one that separates packets with local and remote source IP addresses, and another that hashes source IP addresses and distributes packets between a configurable number of sandboxes. In each case, a catch-all sandbox is created for non-IP packets. This ensures that packets from one flow group cannot effect the processing of other packets without first escaping the sandbox.

To reduce the impact of an attacker gaining full control of the sandbox, we also implemented limits on packet count, and on the time between resets of group sandboxes (temporal compartmentalization). These limits reduce the window during which a successful attacker can manipulate the display of packet or protocol data (or attempt to escape the sandbox), and the amount of exposed past data. We also implemented vertical compartmentalization in the form of per-protocol sandboxes, where each layer of protocol parsing is dispatched to another sandbox object. In a malicious environment, this allows `tcpdump` to do as much work as possible for the operator without compromising overall integrity. For example, exploits in a higher-level protocol (such as bad ASN.1 in an SNMP packet) cannot obscure Ethernet, IP, or UDP headers. Finally, we defend against denial-of-service attacks involving infinite or long-running loops by implementing a simple work-bounded compartmentalization in which processing a packet is timed out by setting an `alarm()`.

Horizontal compartmentalization enables treating different packets with different trust properties, reducing the impact of attacks from reliably identifiable sources (e.g., remote networks). Vertical compartmentalization allows reliable partial processing of packets that trigger bugs up to the layer where the bug occurs, allowing maximum information to be derived from malicious packets. If an attacker takes control of a compartment without detection, temporal sandboxing limits the the effects to a set of packets or window of time. Work-bounded compartmentalization limits an attacker's ability to halt packet processing through denial of service. In the current prototype, horizontal and vertical compartmentalization are not composable, but could be – with a simple extension.

These compartmentalizations are sufficient to mitigate all but two of the analyzed vulnerabilities. CVE-2003-0194 provides a mechanism for privilege escalation, and CVE-2014-8767 allows an attacker to make arbitrary calls to `gethostbyaddr()`. However, since rights need to be delegated in order for an attack to succeed, we consider this to be a policy rather than mechanism issue.

These features required few changes to the `tcpdump` code base, and the addition of just over 1% new code. Modest source-code rearrangement was required to allow the dissection code to be compiled as a self-contained unit. These changes seem likely to be accepted upstream, as they are a uncontroversial relocation of a dozen or so functions to different files. Horizontal and temporal compartmentalization required fewer than 600 lines of new code, which support for creation and invocation of compartments, with another 150 to set up state compartments before invoking print routines. Vertical compartmentalization required more extensive modifications, with each packet-dissector function being wrapped to either call the actual dissector or invoke a method in a another compartment. This required wrapping 108 functions with a simple function to call the target function in the next sandbox (if available) or in the current sandbox. The infrastructure to declare dissectors using the `cheri_ccall` calling convention is showing in Figure 9 with the implementation of the wrapper function shown in Figure 10. Inside the compartment, `ip_print` is declared `cheri_ccallee` allowing it to be called directly with only callee side register clearing overheads.

```
extern struct cheri_object cheri_tcpdump;
extern struct cheri_object g_next_object;
#ifdef CHERI_TCPDUMP_INTERNAL
#define CHERI_TCPDUMP_CCALL                                     \
    __attribute__((cheri_ccallee))                              \
    __attribute__((cheri_method_suffix("_cap")))                \
    __attribute__((cheri_method_class(cheri_tcpdump)))
#else
#define CHERI_TCPDUMP_CCALL                                     \
    __attribute__((cheri_ccall))                                \
    __attribute__((cheri_method_suffix("_cap")))                \
    __attribute__((cheri_method_class(cheri_tcpdump)))
#endif

#define ND_DECLARE(name, ...)                                   \
   void name(netdissect_options *, __VA_ARGS__);                \
   CHERI_TCPDUMP_CCALL                                          \
   void _##name(netdissect_options *, __VA_ARGS__)

ND_DECLARE(ip_print, const u_char *, u_int);
```

Fig. 9. Annotated declaration of IP packet dissector

```
void
ip_print(netdissect_options *ndo, const u_char *bp,
    u_int length)
{
    if (!CHERI_OBJECT_ISNULL(g_next_object))
        _ip_print_cap(g_next_object, ndo, bp, length);
    else
        _ip_print(ndo, bp, length);
}
```

Fig. 10. Wrapper for the IP packet dissector

To allow a comparison of only the compartmentalization cost, we produced a version of `tcpdump` where we compiled packet dissection code in pure-capability mode for memory protection, but call the functions directly rather than implementing a sandbox. We have retained Capsicum sandboxing even in the presence of CHERI compartmentalization, as the two techniques provide complementary benefits. Whereas Capsicum limits access by the application to system resources, providing protection to the OS, CHERI limits the scope of attacker behavior within the application.

VI. Performance

Compartmentalization scalability is a key goal of the CHERI architecture – that is, clean scaling as the number of both compartments and their interactions grow. Greater scalability translates into the opportunity for improved resilience, as increased *compartmentalization granularity* improves approximation of the principle of least privilege. To this end, we explore CHERI scalability through a set of micro and macro benchmarks to understand the performance characteristics of our approach in comparison with pure MMU-based techniques.

All benchmarks were run under CheriBSD on the CHERI processor prototype implemented on an Altera FPGA on the Terasic DE4 board. CHERI was clocked at 100MHz, and implemented as an in-order, single-issue, six-stage pipeline. The processor was configured with 16KiB, direct-mapped instruction and data caches, and a 64KiB, 4-way associative L2 cache. It has a 144 entry TLB with 16 associative and 128 direct-mapped entries, each mapping two 4 KiB pages.

A. Micro: Capability Overheads

The CheriBSD kernel and userspace runtime both incur new costs associated with implementing capability support

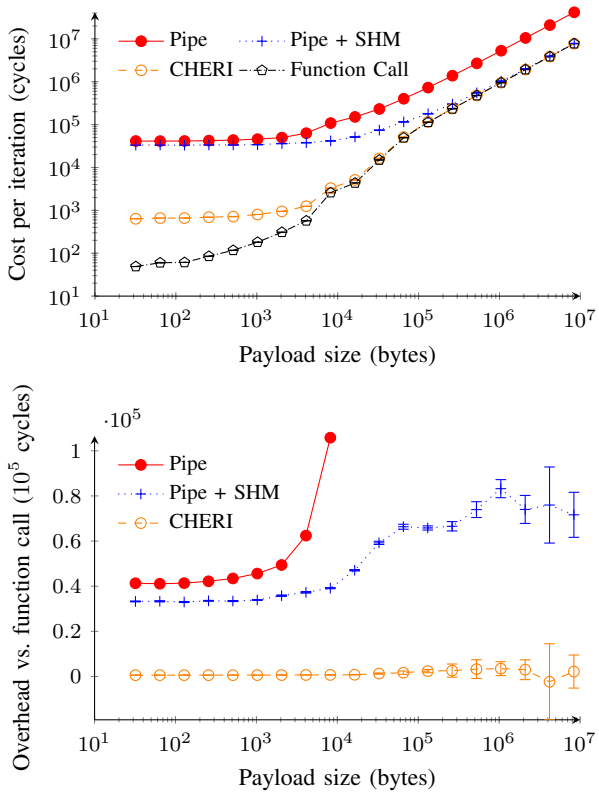

Fig. 11. Micro-benchmark for function call, CHERI CCall/CReturn, pipe RPC, and pipe RPC with shared memory for payload. First graph: log-log scale. Error bars show standard deviation. Second graph: overhead vs. function call; log X axis and linear Y axis. Error bars show square root of sum of standard deviations of method and baseline.

(e.g., saving and restoring larger register files); they also introduce new low-level primitives (e.g., object-capability invocation) intended to provide more scalable alternatives to existing process-based primitives. In prior work, we have explored the baseline costs of CHERI memory protection [57]. This showed an overall overhead of around 5-20% in the common case, with overhead of 50-70% when memory-bandwidth limited with pointers as an overwhelming majority of the data. In the capability ABI, we represent all pointers with the capability mechanism and so would expect to see this overhead added to the cost of domain crossing.

B. Micro: Domain-Crossing Overhead

We created a C program that performs a simulated workload inside a sandbox using four different invocation mechanisms. The workload is a simple `memcpy`, allowing us to measure both the cost of domain transition and the cost of transferring data into and out of the sandbox. The `memcpy` call ensures that the input buffer is read and the output buffer is written to, simulating cache and TLB behaviors of real workloads. The sandboxing mechanisms were as follows:

Function A normal call to a function that performs the `memcpy` and returns. This provides no isolation, but gives a baseline against which to compare other mechanisms.

CHERI We invoke a method on a `libcheri` object that copies data between two buffers passed via capabilities. `libcheri` uses `CCall` to transition between protection

domains; the compiler also generates code to clear unused argument and return-value registers.

Pipe We fork a sandboxed process and send data to it via a UNIX pipe. The sandbox then echos the data back again via the pipe. This IPC model is used in most privilege-separated applications, such as `sshd`.

Pipe + Shared Memory We fork a sandboxed process and communicate with it via a pipe and shared memory. The pipe synchronizes by sending a length argument, while `memcpy` copies data via shared memory. This IPC model is used in larger-scale compartmentalized applications (such as between Chromium's browser process and renderer sandboxes) where bulk data is shared by compartments.

We also implemented the pipe and pipe + SHM cases using a UNIX socket instead of a pipe, but found that pipes perform better under FreeBSD due to VM optimizations to avoid data copying; for clarity we do not include the socket cases in our results. Figure 11 shows the cycle cost of a complete transition into and out of the sandbox for memory copies up to 8MiB (with the arithmetic mean of 50 iterations after excluding outliers caused by timer interrupts and start-up costs).

These results show that CHERI greatly outperforms the other mechanisms, especially for small payload sizes. A function call with zero payload took on average 12 cycles, whereas the equivalent `libcheri` invoke took 632 – an overhead of 620 cycles. Process-based sandboxing was orders of magnitude slower at around 41,000 cycles for the pipe-only case, and 33,000 for pipe + shared memory. At larger sizes, fixed costs are dominated by data-copying costs, and the performance of CHERI, shared memory, and function calls approach each other in the limit, while the pipe case remains five times slower due to extra copies that scale with payload size.

The second graph shows how the absolute overhead scales with data size, after subtracting the function-call baseline. For CHERI this remains roughly constant, while pipe overhead is proportional to the amount of data transferred: it quickly leaves the top of the log-linear graph. Shared-memory IPC initially incurs a high fixed overhead due to synchronization using the pipe, then sees a further increase as the data set begins to span multiple pages: shared memory requires twice the number of TLB entries to map the data in both the parent and the sandbox, causing more TLB misses.

For all but huge data sets, IPC cost dominates when using process isolation. In CHERI, the reverse is true; the time spent executing the `memcpy` dominates for sizes over about 4KiB. For workloads with modest computation inside the sandbox, we would expect CHERI to have acceptable overhead for even smaller data sizes. For process-based compartmentalization, the cost of computation inside the sandbox would have to be significantly greater for the same amortization.

C. Micro: Object Invocation Costs

To better understand the costs of a method invocation with `libcheri`, we ran the benchmark with instruction-level tracing on our CHERI prototype, and analyzed the traces with respect to a variety of metrics including cycle and instruction counts. We took multiple samples, discarding runs in which timer interrupts fired, giving us values for *best case* domain transition. Table VIII shows the instruction count and average cycle costs for different phases of `CCall` and `CReturn`.

TABLE VIII. CHERI OBJECT-CAPABILITY INVOCATION COSTS

Invocation Phase	Software		HW support	
	Inst.	Cyc.	Inst.	Cyc.
caller: Setup call, clear unused argument regs.	22	42	22[c]	42[c]
libcheri: Save callee-save regs, push call frame	30	34	30	34
kernel: Receive trap	13	28[a]	13	28[a]
kernel: Validate CCall args.	94	94	79[d]	79[d]
kernel: Push trusted stack, unseal CCall args.	31	47	31	41
kernel: Clear non-argument registers	38	59	4[b]	4[b]
kernel: Exit kernel	7	9	7	12
sandbox: Set up sandbox	33	59[a]	33	59[a]
sandbox: memcpy (payload)	4	17	4	17
sandbox: Exit sandbox	12	16	12	16
kernel: Receive trap	13	29[a]	13	31[a]
kernel: Validate return capability	7	7	7	7
kernel: Pop trusted stack	26	41	26	41
kernel: Clear non-return registers	54	84	4[b]	4[b]
kernel: Exit kernel	7	7	7	7
libcheri: Pop call frame, restore regs.	28	58[a]	28	52[a]
caller: Back in caller	1	1	1	1
Total	420	632	321	475

[a] Includes overhead of hardware exception (10-15 cycles)
[b] Savings due to hardware clear regs instruction
[c] Compiler does not yet support clear regs instruction so further savings possible
[d] Savings due to hardware `CCall` validation

We observed a large cost in clearing registers (including capabilities) that are not arguments or return values. This is necessary to prevent leaked data or capabilities that might allow attacks across the interface. We hypothesize that further hardware support for clearing registers would reduce this cost. We emulated this by replacing register clearing with no-op instructions – four hypothetical instructions would be required to clear both register files. We found that this directly saved 135 cycles, with further savings of 5-10 cycles likely due to reduced instruction-cache usage.

Another large cost is validating that capability arguments and return values conform with the `CCall` semantics and information-flow policy. We prototyped hardware support for validating the `CCall` arguments and obtained a modest saving of about 15 cycles. Further optimization opportunities exist, but this validation is specific to CheriBSD's compartment memory model, creating a tradeoff between generality and performance.

Other significant costs include saving and restoring callee-save registers (12 general purpose, 11 capability), manipulating the trusted stack, trap overhead (four times for the call and return sequence), and cache and TLB usage. We are investigating ways to further reduce these costs by tuning aspects of our implementation and model (e.g. the number and size of capability registers), but for now err on the side of generality. Nevertheless, we believe that even our unoptimized costs represent acceptable overhead for real applications, as shown in the following sections through macro benchmarks.

D. Macro: `zlib` / `gzip`

Conventional MMU-based compartmentalization techniques (such as Capsicum) rely on the UNIX process model to create isolated compartments, and on UNIX IPC to bridge those compartments for the purposes of protection-domain crossing and data sharing. These techniques introduce three substantial costs: (1) the instantaneous overheads of process

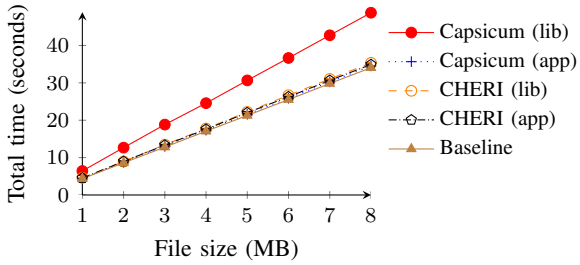

Fig. 12. Compression time for `gzip` with various sandboxing mechanisms

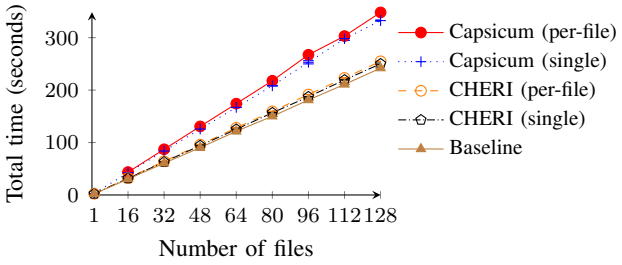

Fig. 13. Sandbox creation overhead: time taken to compress varying numbers of files of size 500,000 bytes with different `zlib` implementations

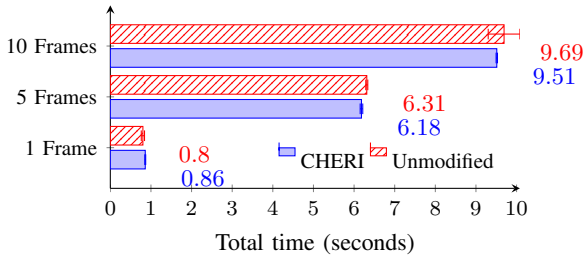

Fig. 14. `gif2png` with unmodified and CHERI `zlib` implementations

creation and destruction to instantiate and destroy compartments; (2) the amortized and indirect costs of additional virtual address spaces, such as TLB contention and kernel-data-structure footprint on the cache; and (3) the cost of protection-domain switching and implied copying (or MMU operations) to pass data between compartments using IPC.

In some cases, these overheads are negligible or amortized. For example, Capsicum allows processes to acquire access to capabilities while operating with ambient authority, and then to enter capability mode without creating an additional process. For simple program structures, such as in `tcpdump`, Capsicum sandboxing adds only a few extra system calls to limit future file-I/O operations and give up ambient authority. However, for other program structures, the cost can be substantial: e.g., if resources cannot be naturally delegated, such as the output of byte streams that have been decompressed within the application, and thus require processing in another compartment. This leads to IPC costs linear on decompressed data size.

Using `zlib` and `gzip` as case studies, we compare two compartmentalization strategies and two compartmentalization technologies to understand when each is most appropriate. The first strategy splits the application, delegating access to I/O file descriptors into the sandbox. The second strategy splits the library, creating a sandbox that implements all library API calls. We implemented both using CHERI in-process compartmentalization, and Capsicum process-based compartmentalization, an approach representative of other OS-based compartmentalization technologies.

Figure 12 compares the performance of these approaches. Three results are very close to the baseline: CHERI, placing the protection boundary in the application or library, and Capsicum, with the boundary in the application. The latter reproduces the results from the Capsicum paper: creating a second process is a small, fixed cost, after which unmodified I/O and compression operations take place. CHERI experiences a small but measurable additional cost on each call to `deflate` within `zlib` due to protection-domain switching overhead. Much of this overhead is due to using memory capabilities to represent all pointers in the sandbox – not because of the larger cache footprint, but because `zlib` performs a number of integer-to-pointer conversions on the critical path. These work without defeating memory safety, but do defeat a number of compiler optimizations in our current implementation.

Process-based library compartmentalization is considerably slower than the baseline due to overheads that scale poorly with data size: all data is read into the ambient process, sent via IPC to the child, compressed, and returned via IPC to the parent to be written to a file. This is extremely inefficient, but is required to maintain current buffer-oriented library APIs.

To investigate sandbox creation costs, we modified the two `zlib` compartmentalizations to use a new sandbox for each compression context (corresponding to a file in the `gzip` program). As shown in Figure 13, process-based sandboxing for `zlib` sees little variation between single- and multiple-sandbox versions. This is because the cost of sandbox creation, while high, remains dominated by IPC cost. In the CHERI case, there is a small constant overhead for each new sandbox, which could be reduced through further optimization.

The advantage of library compartmentalization is that a single investment in developer effort can provide security gains for many applications. We demonstrated this by linking `gif2png` – a simple tool for image format translation that uses `zlib` indirectly via another library – to our modified `zlib`. This required no code changes in the application, and illustrates that our approach encourages code reuse in a security context. Figure 14 compares execution times for the unmodified and compartmentalized versions of `zlib` to convert single-frame (16KB), five-frame (100KB), and ten-frame (200KB) `.gif` files to `.png`. In `gif2png`, performance is dominated by work other than the compression, and users are unlikely to notice the library changes, which are within the margin of error in these tests.

E. Macro: tcpdump

Three factors dominate the cost of sandbox compartmentalization in `tcpdump`: initialization, domain crossing, and reset (for models where sandboxes are periodically reset). To measure the practical overhead, we fed groups of 1000 packets from a generated TCP packet capture file to `tcpdump` while suppressing output, which otherwise dominates performance. We compared a number of sandbox counts ranging from 1 to 128 for IP flow groups (for 2 to 129 total sandboxes). The time to initialize `tcpdump` in each case is shown in Figure 15. The time to process the 1000 packets is shown in Figure 16.

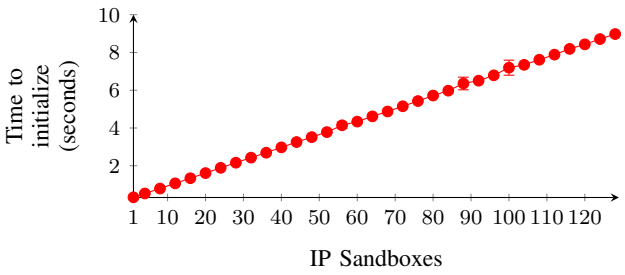

Fig. 15. Time to start tcpdump vs. number of IP flow-group sandboxes

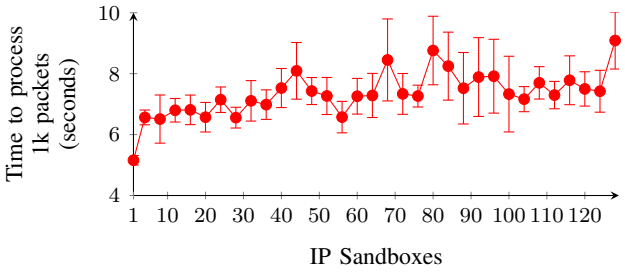

Fig. 16. Time to process 1000 packets vs. number of IP flow-group sandboxes

As expected, startup cost is linear in the number of sandboxes. Packet-processing cost increases steeply from the one-sandbox case to the eight-sandbox case, settling into a pattern of very slow and roughly linear growth – roughly a 1% growth per sandbox at a high statistical significance explaining roughly 15.5% of variability – demonstrating the viability of large numbers of intra-process sandboxes. At 128 sandboxes, `tcpdump` has mapped 1GiB of address space – the entirety of physical memory of our platform. Further `tcpdump` optimization would reduce both sandbox startup time and the per-object memory footprint, with a primary memory overhead being large (and mostly unused) global variables that `libcheri` replicates for all `tcpdump` object instances.

VII. Design-space considerations

Capability systems have a long and rich history, with many points on the design space across hardware, operating systems, and programming languages. CHERI adopts ideas from many of these: hardware grounding to provide strong underlying integrity and fast-path acceleration; a hybrid design to provide source-code and binary compatibility with current software; vulnerability mitigation rather than simply access control; and a focus on programming languages rather than APIs.

Application compartmentalization drove most design choices in CHERI: capabilities are linked to the C language, and tagged memory allows capabilities to replace pointers within existing data structures. This design allows capabilities to flow easily through the system as they are propagated by memory copies and passed implicitly as function arguments and return values. Supervisor intervention is avoided to keep all common capability operations, other than invocation and return, fast. Indirection is explicitly avoided in our RISC design: there are no segment or capability lookup tables as found in classical hardware or microkernels.

This makes using and sharing capabilities easy, but revoking capabilities hard: we must instead rely on stronger software invariants (e.g., address-space non-reuse) and techniques such as information flow control and garbage collection. Tags facilitate these goals: reliable C garbage collection is possible on CHERI, but this means that applications requiring frequent synchronous revocation, not just frequent sharing, may experience greater overhead. We believe that the benefits of tight language integration substantially outweigh the costs of more subtle security semantics for memory – but it remains to be seen what implications this will have for larger code bases. A key concern will be "leaked capabilities" – either application-level programming errors in which data and objects are accidentally leaked to callers or callees, or implementation errors in the compiler or memory management.

An early goal was for CHERI to support a single-cycle domain-transition model via a dedicated instruction, reducing its cost to that of an ordinary function call; this goal was not met, although a multiple orders-of-magnitude reduction was accomplished. On reflection, the goal was naive: as our analysis shows, much of the remaining domain-transition overhead, relative to more porous function calls, lies in the cache footprint of additional operations required for security.

Further, it became clear that there were a huge variety of security models that could be built over CHERI memory protection, spanning asynchronous and synchronous designs, with or without notions of TCB-supported exception recovery, and linked in various ways to memory safety models (e.g., garbage collection). Our current hardware-assisted exception-based domain switch allows the TCB to implement complex behaviors not suitable for a RISC pipeline (e.g., trusted-stack manipulation). While some of that behavior could be shifted to caller and callee contexts, other elements cannot: restricting the flow of local capabilities and trusted-stack manipulation protects the object model itself, and are not just defensive behaviors for mutually distrusting compartments.

If we were to start again from scratch, there are choices that we might make differently, but these are largely surface aspects; for example, separating general-purpose and capability registers reduced ABI change, but came at a cost to cache footprint. The fundamental choices to retain an MMU to support current software, tagged capabilities to allow language integration, tight compiler integration to avoid RPC-like stubs, and a software-defined security model over a memory-protection substrate, have proved transformative foundations.

VIII. Future Work

We have demonstrated that the CHERI ISA and software stack can act as the foundations for a both more programmer-friendly and more scalable software compartmentalization platform. However, our current prototypes scratch only the surface of the possible explorations that could be performed, and we hope to continue this work in the following ways.

We describe a simple userspace memory model that provides safe communication between compartments with mutual distrust. Previous focus has been spatial integrity rather than temporal protection, leaving opportunities for programmer error if memory is freed too quickly by the larger application. Tagged capabilities offer a straightforward solution: accurate garbage collection is a real possibility. In CheriBSD, we chose to support tagged capabilities only for anonymous (swap-backed) memory, retaining current filesystems and avoiding

temporal safety problems associated with stale address-space assumptions for persistent capabilities. However, mechanisms exist within the CheriBSD VM subsystem to implement more complex models, such as persistent stores in which tags are maintained across application crash or system reboot.

Many questions remain open regarding how to develop software to best benefit from the CHERI architecture. Automatic tools to implement compartmentalization are far more feasible with a CHERI-like substrate, where introducing security boundaries does not require substantial restructuring of code to employ message passing – as is the case with conventional process-based privilege separation. We have focused on hybridization with current software designs, exploiting the retained MMU to support existing operating systems and programming models. However, the CHERI ISA can support many other models – e.g., deemphasizing the MMU to implement a single-address-space capability model.

This paper informally summarizes some of the properties required for CHERI to be meaningfully trustworthy – relating to correctness of the hardware specification, security and system integrity, compartmentalization, and so on. We are conducting formal analyses of the hardware and to some extent low-level software, having developed the infrastructure to facilitate such analyses and their hierarchical closure.

IX. Related work

Our CHERI hardware-software security model draws on a long history of work on the principles of computer security, access control, capability systems, operating systems, and programming languages [43], [4]. Early access-control systems focused on discretionary and mandatory access control, employing security attributes or labels to control information and control flow to protect confidentiality [7], integrity [9], and availability. Multics [16] was an early testbed for many of these ideas, with detailed investigations [8], [26] of implications of the security techniques. During the 1990s, OS-centered access control transitioned from user-focused policies [24] to vulnerability mitigation [5], with systems such as Linux [32], FreeBSD, and Mac OS X/iOS [52] employing access controls to limit attacker rights in increasingly single-user systems.

Capability systems also have a long history [17], [31], with hardware-software systems such as the tagged and typed-object PSOS design [40] and the CAP [55] implementation, and through the 1990s and 2000s transitioning first to operating systems such as Hydra [58], Mach [2], EROS [45], and SeL4 [28], and later programming languages such as E [35], [48], Joe-E [34], and Caja [36]. This transition from capabilities referring to low-level, fixed-function objects to a more general compartmentalization model, groundwork laid by systems such as Hydra and PSOS, and building on notions of "protected subsystem" from earlier designs, becomes the foundation for object-capability systems in which interposition at an object level becomes a key means of supporting higher-level policies. Hybrid capability systems represent an effort to provide an incremental adoption path for capability-system benefits in conventional system designs, and are epitomized by systems such as Capsicum [53] and Joe-E [34].

CheriBSD's object-capability model is strongly influenced by HYDRA: our trusted stack records synchronous object invocations able to pass typed capabilities between protection domains within a thread of execution. However, whereas CHERI's capabilities are represented directly in the ISA, HYDRA relied on an MMU-based process model with capabilities implemented in the kernel. CheriBSD invocation requires explicit type checking and unsealing of argument objects by the callee (i.e., no implicit amplification).

CHERI is also strongly influenced by M-Machine [13], which provided tagged memory in support of fine-grained memory capabilities. Whereas M-Machine implemented an asynchronous model (reasonably described as *secure closures*, combining code and data references in entry and return capabilities, allowing a single-instruction call/return mechanism), CheriBSD implements *secure object invocation* based on a TCB-maintained reliable return stack, and separate code and data capabilities. CHERI's exception-handler-based approach can support a range of software-defined models including the M-Machine model. Unlike M-Machine, CHERI maintains source-code and binary compatibility with current software stacks through retention of a conventional MMU, process model, C language, and interoperable ABIs.

Hardware foundations for security have co-evolved with both access-control and capability-system techniques. Extending then-contemporary user/supervisor splits, Multics promoted a more granular ring-based model [44] and fine-grained separation via independent segments, as did many successor systems such as the Intel x86 architecture (until removal in recent 64-bit extensions). These protection mechanisms were deemphasized through the 1990s, but there has been a recent resurgence due to interest in full-system virtualization, system management modes, and hardware-supported security models.

ARM's TrustZone [3] and Intel's Software Guard Extension (SGX) [22] also address a form of compartmentalized software design in which commodity operating systems are considered insufficiently trustworthy to host critical security functions such as authentication and financial transactions. They respectively provide support for an independent security-focused kernel or application elements alongside the current operating system, and a hardware-supported model for "application enclaves" in which components of applications running on top of the conventional OS are protected from its interference. CHERI's fine-grained compartmentalization could be viewed as complementary: TrustZone- and SGX-protected software elements would benefit from fine-grained internal compartmentalization to mitigate attacks from the untrustworthy software platform.

These features were deemphasized in favor of a more coarse-grained paging model used by UNIX-like systems through the 1990s. More recently, interest in stronger memory safety within processes has grown, including software-based C-language-based systems such as Cyclone [23], Softbound [38], CCured [39], low-fat pointers [29], and Control-Flow Integrity (CFI) [1]. Hardware solutions have also been proposed: HardBound [18], and more recently, Intel Memory Protection eXtensions (MPX) [21] have attempted to accelerate fat-pointer performance. However, these systems focus on exploit mitigation rather than compartmentalization.

Software and hardware systems have been used to explore compartmentalization efficiency. Software transformation approaches such as Software Fault Isolation (SFI) [49] and Google NaCl [59] have focused on strong and efficient isola-

tion without hardware support, rather than catering to many tightly interlinked compartments. In hardware, Mondriaan investigated an access-control-centered approach based on a TLB/MMU page-table-like mechanism to represent in-address-space security domains, including running an adaptation of Linux [56]. CRASH-SAFE has more recently explored flexible, software-defined, tagged security models, often grounded in information flow, based on clean-slate ISA approaches [14]. Hypervisors have been used to provide contained execution environments [37], and Dune utilizes hardware virtualization features to accelerate intra-process isolation [6].

Software compartmentalization to mitigate vulnerabilities was first proposed by Karger [25] using capability-system approaches, and later popularized using sandboxed UNIX processes by Provos [41] and Kilpatrick [27]. It has since become widespread in systems such as FreeBSD and Mac OS X [52], as well as applications such as Chromium [42]. Automated techniques for privilege separation, as well as optimization of its primitives, has been the focus of systems such as Privtrans [12], Wedge [10], and Capsicum [53].

X. Conclusion

We have described extensions to the CHERI Instruction-Set Architecture that enable the building of scalable and highly compatible object-capability systems. Building on CHERI's hybrid capability-model approach, and in contrast to historic hardware capability-system designs, we demonstrate that a fine-grained in-address-space protection model can be the foundation for efficient protection-domain switching that retains support for the UNIX process model and C-language software stacks. As a result, the CHERI object-capability model is incrementally deployable to current code bases that have experienced long histories of vulnerabilities – and also adoptable in "less than clean-slate" processor designs.

Our hardware-software prototype enables an integrated approach to security design and evaluation that explores transformative scalability improvements – multiple orders-of-magnitude reductions in domain-transition costs – through realistic research artifacts. We demonstrate that our choice of execution substrate eases many of the challenges of previous software compartmentalization. For example, library compartmentalization (which had previously challenged OS-based approaches) with CHERI allows binary-compatible improvements in security and robustness, without modifying containing applications. We have open-sourced our hardware and software designs to support greater experimental reproducibility, as well as to encourage further exploration of our approach.

XI. Acknowledgments

We thank our colleagues Ross Anderson, Ruslan Bukin, Gregory Chadwick, Steve Hand, Alexandre Joannou, Chris Kitching, Wojciech Koszek, Bob Laddaga, Patrick Lincoln, Ilias Marinos, A Theodore Markettos, Ed Maste, Andrew W. Moore, Alan Mujumdar, Prashanth Mundkur, Colin Rothwell, Philip Paeps, Jeunese Payne, Hassen Saidi, Howie Shrobe, and Bjoern Zeeb, our anonymous reviewers, and shepherd Frank Piessens, for their feedback and assistance. This work is part of the CTSRD and MRC2 projects sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237 and FA8750-11-C-0249. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government. We acknowledge the EPSRC REMS Programme Grant [EP/K008528/1], Isaac Newton Trust, UK Higher Education Innovation Fund (HEIF), Thales E-Security, and Google, Inc.

References

[1] Abadi, M., Budiu, M., Úlfar Erlingsson, and Ligatti, J. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the 12th ACM conference on Computer and Communications Security* (2005), ACM, pp. 340–353.

[2] Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: A New Kernel Foundation for UNIX Development. Tech. rep., Computer Science Department, Carnegie Mellon University, August 1986.

[3] Alves, T., and Felton, D. TrustZone: Integrated hardware and software security. *Information Quarterly 3*, 4 (2004).

[4] Anderson, J. Computer security technology planning study. Tech. Rep. ESD-TR-73-51, U.S. Air Force Electronic Systems Division, October 1972. (Two volumes).

[5] Badger, L., Sterne, D., Sherman, D., Walker, K., and Haghighat, S. Practical domain and type enforcement for Unix. In *Proceedings of the 1995 Symposium on Security and Privacy* (May 1995), IEEE.

[6] Belay, A., Bittau, A., Mashtizadeh, A., Terei, D., Mazières, D., and Kozyrakis, C. Dune: safe user-level access to privileged CPU features. In *Proceedings of the 10th Conference on Operating Systems Design and Implementation* (2012), USENIX.

[7] Bell, D., and Padula, L. L. Secure computer systems : Volume I – mathematical foundations; volume II – a mathematical model; volume III – a refinement of the mathematical model. Tech. Rep. MTR-2547 (three volumes), The Mitre Corporation, Bedford, Massachusetts, March–December 1973.

[8] Bell, D., and Padula, L. L. Secure computer system: Unified exposition and Multics interpretation. Tech. Rep. ESD-TR-75-306, The Mitre Corporation, Bedford, Massachusetts, March 1976.

[9] Biba, K. Integrity considerations for secure computer systems. Tech. Rep. MTR 3153, The Mitre Corporation, Bedford, Massachusetts, June 1975. Also available from USAF Electronic Systems Division, Bedford, Massachusetts, as ESD-TR-76-372, April 1977.

[10] Bittau, A., Marchenko, P., Handley, M., and Karp, B. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation* (2008), USENIX.

[11] Boebert, W., and Kain, R. A practical alternative to hierarchical integrity policies. In *Proceedings of the Eighth DoD/NBS Computer Security Initiative Conference* (1–3 October 1985).

[12] Brumley, D., and Song, D. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th USENIX Security Symposium* (2004), USENIX.

[13] Carter, N. P., Keckler, S. W., and Dally, W. J. Hardware support for fast capability-based addressing. *SIGPLAN Not. 29*, 11 (Nov. 1994), 319–327.

[14] Chiricescu, S., DeHon, A., Demange, D., Iyer, S., Kliger, A., Morrisett, G., Pierce, B. C., Reubenstein, H., Smith, J. M., Sullivan, G. T., Thomas, A., Tov, J., White, C. M., and Wittenberg, D. SAFE: A clean-slate architecture for secure systems. In *Proceedings of the IEEE International Conference on Technologies for Homeland Security* (Nov. 2013).

[15] Chisnall, D., Rothwell, C., Davis, B., Watson, R. N., Woodruff, J., Vadera, M., Moore, S. W., Neumann, P. G., and Roe, M. Beyond the PDP-11: Processor support for a memory-safe C abstract machine. In *Proceedings of the 20th Architectural Support for Programming Languages and Operating Systems* (2015), ACM.

[16] Corbató, F. J., and Vyssotsky, V. A. Introduction and overview of the Multics system. In *AFIPS '65 (Fall, part I): Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I* (New York, NY, USA, 1965), ACM, pp. 185–196.

[17] DENNIS, J. B., AND VAN HORN, E. C. Programming semantics for multiprogrammed computations. *Commun. ACM 9*, 3 (1966), 143–155.

[18] DEVIETTI, J., BLUNDELL, C., MARTIN, M. M. K., AND ZDANCEWIC, S. Hardbound: architectural support for spatial safety of the C programming language. *SIGARCH Comput. Archit. News 36*, 1 (Mar. 2008), 103–114.

[19] GONG, L., MUELLER, M., PRAFULLCHANDRA, H., AND SCHEMERS, R. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *Proceedings of the Symposium on Internet Technologies and Systems* (December 1997), USENIX.

[20] HEINRICH, J. *MIPS R4000 Microprocessor User's Manual (Second Edition)*. MIPS Technologies, Inc, 1994.

[21] INTEL PLC. Introduction to Intel memory protection extensions. http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions, July 2013.

[22] INTEL PLC. Intel Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, October 2014.

[23] JIM, T., MORRISETT, J. G., GROSSMAN, D., HICKS, M. W., CHENEY, J., AND WANG, Y. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference* (2002), pp. 275–288.

[24] KAMP, P., AND WATSON, R. N. M. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference* (2000).

[25] KARGER, P. Limiting the damage potential of discretionary Trojan horses. In *Proceedings of the 1987 Symposium on Security and Privacy* (April 1987), IEEE.

[26] KARGER, P., AND SCHELL, R. Multics security evaluation: Vulnerability analysis. In *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC), Classic Papers section* (Las Vegas, Nevada, December 2002). Originally available as U.S. Air Force report ESD-TR-74-193, Vol. II, Hanscomb Air Force Base, Massachusetts.

[27] KILPATRICK, D. Privman: A Library for Partitioning Applications. In *Proceedings of 2003 USENIX Annual Technical Conference* (2003).

[28] KLEIN, G., ANDRONICK, J., ELPHINSTONE, K., HEISER, G., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: Formal verification of an operating-system kernel. *Commun. ACM 53* (June 2009), 107–115.

[29] KWON, A., DHAWAN, U., SMITH, J. M., KNIGHT, JR., T. F., AND DEHON, A. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *20th ACM Conference on Computer and Communications Security* (November 2013).

[30] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and runtime optimization* (2004), IEEE.

[31] LEVY, H. M. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[32] LOSCOCCO, P. A., AND SMALLEY, S. D. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the USENIX Annual Technical Conference* (June 2001).

[33] MCKUSICK, M. K., NEVILLE-NEIL, G. V., AND WATSON, R. N. M. *The Design and Implementation of the FreeBSD Operating System*. Pearson, 2014.

[34] METTLER, A., WAGNER, D., AND CLOSE, T. Joe-E: A Security-Oriented Subset of Java. In *NDSS 2010: Proceedings of the Network and Distributed System Security Symposium* (2010).

[35] MILLER, M. S. *Robust composition: towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, Baltimore, MD, USA, 2006.

[36] MILLER, M. S., SAMUEL, M., LAURIE, B., AWAD, I., AND STAY, M. Caja: Safe active content in sanitized javascript, May 2008. http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf.

[37] MURRAY, D. G., AND HAND, S. Privilege Separation Made Easy. In *Proceedings of the ACM SIGOPS European Workshop on System Security (EUROSEC)* (2008), ACM.

[38] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M. K., AND ZDANCEWIC, S. SoftBound: highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation* (2009), ACM.

[39] NECULA, G. C., MCPEAK, S., AND WEIMER, W. CCured: Type-safe retrofitting of legacy code. *ACM SIGPLAN Notices 37*, 1 (2002), 128–139.

[40] NEUMANN, P., BOYER, R., FEIERTAG, R., LEVITT, K., AND ROBINSON, L. A Provably Secure Operating System: The system, its applications, and proofs. Tech. rep., Computer Science Laboratory, SRI International, May 1980. 2nd edition, Report CSL-116.

[41] PROVOS, N., FRIEDL, M., AND HONEYMAN, P. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium* (2003), USENIX.

[42] REIS, C., AND GRIBBLE, S. D. Isolating web programs in modern browser architectures. In *EuroSys '09: Proceedings of the 4th ACM European Conference on Computer Systems* (2009), ACM.

[43] SALTZER, J. Protection and the control of information sharing in Multics. *Commun. ACM 17*, 7 (July 1974), 388–402.

[44] SCHROEDER, M., AND SALTZER, J. A hardware architecture for implementing protection rings. *Commun. ACM 15*, 3 (March 1972).

[45] SHAPIRO, J., SMITH, J., AND FARBER, D. EROS: a fast capability system. In *Proceedings of the seventeenth ACM Symposium on Operating Systems Principles* (Dec 1999).

[46] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. Eternal war in memory. In *IEEE Symposium on Security and Privacy* (2013).

[47] THE MITRE CORPORATION. Common Vulnerabilities and Exposures List. https://cve.mitre.org, Feb 2015.

[48] WAGNER, D., AND TRIBBLE, D. A security analysis of the combex darpabrowser architecture, March 2002. http://www.combex.com/papers/darpa-review/security-review.pdf.

[49] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. U. L. Efficient software-based fault isolation. In *Proceedings of the 14th Symposium on Operating Systems Principles* (1993), ACM.

[50] WANG, Z., AND LEE, R. Covert and side channels due to processor architecture. In *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual* (Dec 2006), pp. 473–482.

[51] WATSON, R. N., WOODRUFF, J., CHISNALL, D., DAVIS, B., KOSZEK, W., MARKETTOS, A. T., MOORE, S. W., MURDOCH, S. J., NEUMANN, P. G., NORTON, R., AND ROE, M. Bluespec Extensible RISC Implementation: BERI Hardware reference. Tech. Rep. UCAM-CL-TR-852, University of Cambridge, Computer Laboratory, Apr. 2014.

[52] WATSON, R. N. M. A decade of OS access-control extensibility. *Commun. ACM 56*, 2 (Feb. 2013).

[53] WATSON, R. N. M., ANDERSON, J., LAURIE, B., AND KENNAWAY, K. Capsicum: Practical capabilities for Unix. In *Proceedings of the 19th USENIX Security Symposium* (August 2010), USENIX.

[54] WATSON, R. N. M., NEUMANN, P. G., WOODRUFF, J., ANDERSON, J., CHISNALL, D., DAVIS, B., LAURIE, B., MOORE, S. W., MURDOCH, S. J., AND ROE, M. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-set architecture. Tech. Rep. UCAM-CL-TR-864, University of Cambridge, Computer Laboratory, Dec. 2014.

[55] WILKES, M., AND NEEDHAM, R. *The Cambridge CAP computer and its operating system*. Elsevier North Holland, New York, 1979.

[56] WITCHEL, E., CATES, J., AND ASANOVIĆ, K. Mondrian memory protection. *ACM SIGPLAN Notices 37*, 10 (2002), 304–316.

[57] WOODRUFF, J., WATSON, R. N. M., CHISNALL, D., MOORE, S. W., ANDERSON, J., DAVIS, B., LAURIE, B., NEUMANN, P. G., NORTON, R., AND ROE, M. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 41st International Symposium on Computer Architecture* (June 2014).

[58] WULF, W., COHEN, E., CORWIN, W., JONES, A., LEVIN, R., PIERSON, C., AND POLLACK, F. HYDRA: the kernel of a multiprocessor operating system. *Commun. ACM 17*, 6 (1974), 337–345.

[59] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native Client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th Symposium on Security and Privacy* (2009), IEEE.