IEEE Micro Special Issue on Security

Submission title

Fast Protection-Domain Crossing in the CHERI Capability-System Architecture

Submission contact information

Please direct all correspondence to:

Dr Robert N. M. Watson
University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Voice: +44 (0)1223 763 569
Fax: 44 (0)1223 334 678
E-mail: `robert.watson@cl.cam.ac.uk`

Abstract

Capability Hardware Enhanced RISC Instructions (CHERI) supplement the conventional Memory Management Unit (MMU) with Instruction-Set Architecture (ISA) extensions that implement an in-address-space capability-system model. CHERI capabilities can also underpin a hardware-software object-capability model for scalable application compartmentalization that can mitigate broader classes of attack. This paper describes ISA additions to CHERI that support fast protection-domain switching, not only in terms of low cycle count, but also efficient memory sharing with mutual distrust. We propose ISA support for sealed capabilities, hardware-assisted checking during protection-domain switching, a lightweight capability flow-control model, and fast register clearing – while retaining the flexibility of a software-defined protection-domain transition model. We validate this approach through a full-system experimental design including ISA extensions, FPGA prototype (implemented in Bluespec SystemVerilog), and software stack including OS (based on FreeBSD), compiler (based on LLVM), software compartmentalization model, and open-source applications.

Fast Protection-Domain Crossing in the CHERI Capability-System Architecture

Robert N. M. Watson[†], Robert Norton[†], Jonathan Woodruff[†], Alexandre Joannou[†], Simon W. Moore[†], Peter G. Neumann[‡], Jonathan Anderson[§], David Chisnall[†], Nirav Dave[*], Brooks Davis[‡], Khilan Gudka[†], Ben Laurie[*], A. Theodore Markettos[†], Ed Maste[†], Steven J. Murdoch[¶], Michael Roe[†], Colin Rothwell[†], Stacey Son[†], and Munraj Vadera[†]

[†]University of Cambridge [‡]SRI International [*]Google Inc. [§]Memorial University [¶]University College London

1 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) and *software compartmentalization* (which limits privileges and further attack surfaces available to attackers [5, 11, 15]). In compartmentalization, applications are decomposed into isolated ("sandboxed") components that are granted only selected access to system and application resources. Unlike exploit mitigation, compartmentalization does not depend on knowledge of specific attack vectors, and is resistant to an arms race as attack and defense co-evolve.

Compartmentalization relies on two underlying trustworthy primitives, typically provided through a blend of hardware and software: *strong isolation*, often implemented using Operating-System (OS) process models grounded in virtual memory, and *controlled communication*, implemented as Inter-Process Communication (IPC) between processes. These primitives were designed for coarse-grained isolation – e.g., whole applications or even virtual machines; they limit *compartmentalization scalability* in the number of domains, rate of domain switches, and degree of memory sharing. This prevents use of more granular decompositions in larger, security-sensitive applications such as OpenSSH [11] and Chromium [12].

The Capability Hardware Enhanced RISC Instructions (CHERI) ISA is a set of incrementally adoptable architectural extensions for scalable, in-address-space memory protection via a *hybrid capability model* [19, 2, 16]. CHERI supplements the conventional Memory Management Unit (MMU) used to implement virtual-memory-based processes with an in-process, compiler-directed, fine-grained, capability-based memory-protection model. CHERI capabilities are used by the compiler to implement strong protection for pointers, and are saved/restored during context switches between domains at a low fixed cost, rather than relying on associative caches such as the MMU's Translation Lookaside Buffer (TLB) that must be flushed and reloaded from in-memory tables. CHERI capabilities optimize sharing by allowing cheap delegation and avoiding aliasing problems experienced by TLBs as memory sharing increases. These properties are critical to scaling up intra-application compartmentalization that is characterized by frequent domain crossings and extensive memory sharing.

We have used CHERI's ISA facilities as a foundation to build a software *object-capability model* supporting orders of magnitude greater compartmen-

talization performance, and hence granularity, than current designs. 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 FPGA-based processor prototype with *sealed capabilities*, *hardware-accelerated object invocation*, and *fast register clearing*, and the CHERI software stack (LLVM compiler [7] and FreeBSD OS [9]) with a *domain-transition calling convention* and a *userspace object-capability model.* While CHERI learns from prior capability systems, such as HYDRA [20] and the M-Machine [1], we focus on deploying protection with current C-language Trusted Computing Bases (TCBs), composing both capability-system and MMU-based virtual-memory models.

In this paper, we describe CHERI's hybrid capability-system model, which enables fast domain switching, and also efficient memory sharing between mutually distrusting domains, while retaining flexibility through a software-defined and hardware-enforced compartmentalization model. We present refinements to our CHERI object-capability mechanism, published in the *2015 IEEE Symposium on Security and Privacy* [16], placing greater focus on architectural and microarchitectural performance. We introduce support for fast clearing of general-purpose and capability registers to efficiently prevent leaks. We also present a cycle-level analysis of the CHERI domain-switch mechanism, exploring tradeoffs between hardware optimization and software flexibility, and of the impact on microarchitectural elements such as TLB and cache misses.

2 Approach

Aimed at C-Language TCBs, CHERI extends conventional MMU-based hardware and software protection with two incrementally deployable capability-based techniques implemented by the ISA, OS, and compiler:

- **Memory capabilities** replace pointers within address spaces, mitigating memory-based attacks [19, 2].
- **Object capabilities** (implemented using memory capabilities) support scalable software compartmentalization [16].

Capability systems are hardware, software, or distributed systems designed to implement the *principle of least privilege* [3, 13]. *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.

CHERI capabilities extend virtual addresses to protect language-level pointers, offering greater integrity via tags, bounds for spatial protection, and permissions to limit use. CHERI capabilities are similar to *fat pointers*, but also have strong monotonicity properties: Pointer manipulations can only maintain or decrease rights associated with a pointer, not increase them. For example, a pointer whose bounds have been narrowed to a specific memory allocation can neither be used for memory access outside of its bounds, nor be transformed into one that can. Capability integrity is enforced in memory via tags – attempts to overwrite some or all of a pointer in memory will atomically clear the tag. CHERI's capability mechanism provides fine-grained, efficient, and effective protection for in-address-space memory (e.g., heap or stack allocations). Unlike many historic "pure" hardware capability systems [8], CHERI's hybrid capability-system architecture retains a conventional MMU, supporting a broad range of software models as illustrated in Figure 1 – and strong compatibility with contemporary C-language software.

In *software compartmentalization* (a.k.a. *privilege separation*), vulnerabilities are mitigated by decomposing applications into isolated components – each granted only the rights it requires to operate [5, 11]. For example, in conventional process-based compartmentalization, `gunzip` decompression can be executed in a sandbox that has been delegated only capabilities for the files being read from and written to. A successful exploit in the decompression code will yield only those limited rights, requiring the attacker to find and exploit further vulnerabilities. *Compartmentalization granularity* describes the degree of program decomposition. Fine-grained compartmentalization improves mitigation by virtue of the princi-

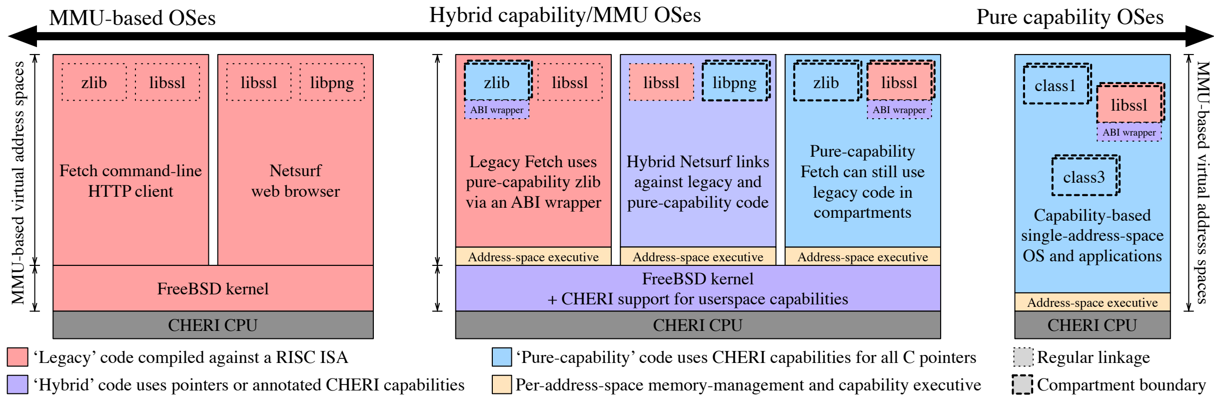


Figure 1: CHERI supports a spectrum of hardware-software architectures.

ple of least privilege: attackers must exploit more vulnerabilities to gain rights in the target system – meaning that improving the performance and scalability of compartmentalization can directly support improvements to software security.

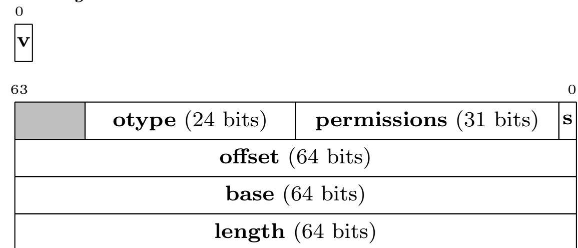
Capability models prove particularly useful in implementing compartmentalization, as they allow programs to easily control what rights are delegated to compartments, and to configure sets of compartments with diverse trust relationships [10, 20, 6, 15]. *Object-capability systems* blend object-oriented OS or programming-language facilities with capabilities to protect application-defined objects. Object encapsulation and interposition then allow programmers to express a range of security policies.

As with MMU-based memory protection, CHERI capabilities can be used to construct a software-defined (but hardware-supported) object-capability model based on isolation and controlled communication. The clean separation of policy and mechanism in object-capability systems aligns elegantly with the RISC (Reduced Instruction Set Computer) philosophy: with protection "fast paths" in hardware, policy definition is left to the OS, compiler, and application. The resulting hardware-software security model can efficiently implement diverse security policies including hierarchical models (e.g., sandboxing) and non-hierarchical models (e.g., mutually distrusting components).

3 Architecture

We begin by briefly describing CHERI memory protection. CHERI extends the 64-bit MIPS ISA with compiler-managed, capability-based, intra-address-space memory protection. Although prototyped with respect to the MIPS ISA, the CHERI approach would likely also apply cleanly to other RISC ISAs.

To implement strong pointer protection using capabilities, CHERI supplements the general-purpose MIPS register file with a *capability register file*. *Capability instructions* allow 256-bit capabilities to be loaded and stored from memory, inspected and manipulated (e.g., to get or set the bounds), dereferenced via load and store instructions, and to be the target of jump and branch instructions. *Capability permissions* control what operations can be performed via a capability. Access via a capability is subject to *tag* validity, relocation relative to its *base* and *offset*, and bounds checking relative to its base and *length*:



(**v** = 1-bit validity **tag**, **s** = 1-bit **sealed**)

Most capability registers are available to compiler and OS-defined Application Binary Interfaces (ABIs), but certain registers are reserved in the ISA. The *program-counter capability* (`$pcc`) extends the MIPS *program counter* (`$pc`) to constrain code execution, and the *exception program counter* (`$epc`) is extended to be the *exception program-counter capability* (`$epcc`). For compatibility, the *default data capability* (`$ddc`) interposes on (or blocks) conventional MIPS loads and stores. Two special capabilities are available during exception handling: the *kernel code capability* (`$kcc`) and *kernel data capability* (`$kdc`).

Guarded manipulation implements *monotonicity*: instructions cannot increase the rights associated with a capability. *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 memory accessible to executing code is the transitive closure of capabilities in its capability register file, and any capabilities reachable through those capabilities. At reset, full capabilities are granted to the boot environment, from which point they may be *delegated* and *refined* from firmware to OS kernel, OS kernel to userspace, and then within user compartments.

3.1 Object-capability extensions

A CHERI-based object-capability model could be constructed without further ISA extensions: 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 controlled communication with 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, permitting object-capability references to be embedded within C data structures.
- To keep important programmer- and compiler-defined paths in userspace, avoiding system calls for permission and type checks.
- To avoid the kernel needing to maintain parallel structures (e.g., tables of objects that must be searched) to implement encapsulation.
- To avoid the need to expose conventional kernel system calls to sandboxed userspace code, reducing the attack surface.
- To efficiently clear non-argument/return-value registers during domain transition.
- To limit capability propagation to reduce the cost of (and need for) garbage collection.

We therefore implement extensions to CHERI memory protection: *sealed capabilities* with *object types*, *capability invocation and return*, instructions for permission and type checking, instructions to clear capability and general-purpose registers, and *capability flow-control* permissions limiting propagation.

3.2 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 an in-address-space protection-domain switch. Object capabilities implement *encapsulation*, restricting 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 an object's 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 in the ISA by a **sealed** bit in the capability. Sealed capabilities are immutable: any attempt to manipulate a field of a sealed capability will throw an exception. The **sealed** bit also prohibits loads (including instruction fetch) and stores. They are otherwise treated as ordinary capabilities with respect to the capability register file and instruction set – and appear as pointers within the programming language.

Sealed code and data capabilities are linked by the **otype** field, a software-defined *object type* that must be identical for a pair of code and data capabilities to be accepted for joint invocation. Capabilities are sealed using the `CSeal` instruction, which accepts two capability-register operands: the code or data memory capability to be sealed, and a second capability with the `Permit_Seal` permission set. The virtual address of a capability with `Permit_Seal` set is treated as a type. Object types prevent instance data from being used with the wrong class.

3.3 Object-capability invocation

Object-capability invocation is implemented via the instructions `CCall`, which invokes a sealed capability pair, and `CReturn`, which returns to the invoking context. 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" – requiring a reliable return to the caller) and asynchronous ("message passing" or "closure") semantics.

`CCall` performs hardware checks (for sealing, suitable permissions, and matching types), selecting an exception vector based on their results. `CReturn` triggers a software exception handler without checks, and may be eschewed if `CCall` is used as an asynchronous message-send primitive. The `CCall` and `CReturn` mechanisms described by the ISA are not sufficient to implement secure protection-domain transition: the software runtime (including the supervisor and userspace runtime) must ensure that memory allocation and capability distribution implement any required isolation, and that register files have been flushed of sensitive data or rights prior to transition.

`CUnseal` 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. It can also be used by a userspace class to unseal other argument objects.

3.4 Capability flow control

CHERI does not prevent use-after-free or other temporal safety violations in hardware; these are controlled by program, language, or run-time mechanisms – e.g., software invariants or garbage collection. When executing within a single protection domain, rapid memory reuse does not constitute a vulnerability in the model. When memory is passed between protection domains, memory reuse could lead to vulnerability.

To assist the software security model in addressing temporal issues, we have extended the CHERI ISA with a 2-bit capability flow-control model that marks capabilities as either global or local. *Global capabilities*, identified by the **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 **Permit_Store_Local** permission set. CheriBSD marks heap references as global, and stack references as local, preventing sharing of stack memory between protection domains. The global/local mechanism restricts only the flow of capabilities, not data.

3.5 Fast register clearing

To prevent information leakage across domain boundaries we must clear unused registers, numerous on RISC processors and increased by capability registers. We introduced `CClearRegs` instructions to efficiently zero up to 16 registers in a single operation.

4 Microarchitecture

The CHERI FPGA soft-core processor implements a capability register file, capability instructions, and tagged physical memory [19]. Minor additions were required to implement compartmentalization-focused ISA extensions, the sealing mechanism, and capability flow control. The costs of these additions were negligible in terms of FPGA resources and critical path, and consumed only a small amount of ISA encoding space.

Efficient implementation of the `CClearRegs` optimization required modification to the register for-

warding logic. We extended the register file with a *zero mask*: a single bit for each architectural register indicates whether a read of that register should return zero. Normal writes to a register clear the bit. Should `CClearRegs` fail to commit, the original register values become readable again.

5 Compiler

We modified the LLVM C front end, MIPS back end, and target-independent optimizers to support fine-grained memory protection and our ISA extensions. We modified around 8KLoC, from a total of just under 2MLoC in LLVM/Clang.

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. Substantially more care is required when crossing protection-domain boundaries. Leaking a capability from a caller to a callee (or vice versa) could have serious integrity, confidentiality, and availability implications.

The compiler implements a new calling convention, `CHERI_CCall`, for functions that can be invoked across domains. With knowledge of the function type, only the compiler is aware of which argument and return-value registers are used. Thus it generates 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 other registers.

6 Operating system

We extended CheriBSD, an adaptation of FreeBSD that supports CHERI memory protection, to implement a lightweight, in-process object-capability model. Kernel changes, summarized in Table 1, add roughly 4KLoC to the approximately 13MLoC kernel.

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. The trusted stack is initially empty, with the first thread of the first process executing with ambient authority (global `$pcc` and `$ddc`). On each invocation, `CCall` saves a code and data capability that `CReturn` will use to resume. The caller is responsible for setting the invoked data capability (`$idc`) to a memory region (typically on the caller's stack) that contains everything needed restore state.

`CCall`, illustrated here in pseudocode, checks that the provided sealed code (`$scc`) and data (`$sdc`) capabilities are valid and properly sealed, and have matching types and suitable permissions:

```
/* 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();
```

It also checks that argument capabilities are 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 registers; this could be done by the caller and callee, but clearing here allows both sides to rely on it always happening, avoiding the need to clear

Subsystem	Description
Program start	On `exec()`, user capability registers grant access to the full user address space.
Context switching	Capability registers are saved and restored during thread context switching.
CHERI exceptions	Capability-related exceptions are mapped to a new `SIGPROT` signal for delivery via the UNIX signal mechanism.
Object capabilities	The kernel's `CCall` and `CReturn` exception handlers implement synchronous object-capability invocation using a *trusted stack*.
System calls	Only user program-counter capabilities with the software-defined `PERM_SYSCALL` permission are permitted to make system calls.
Signal delivery	Signal handlers run in a privileged user context, and might choose to unwind the trusted stack, or deliver a language-level exception.

Table 1: CheriBSD kernel changes to support userspace capabilities

registers in both to prevent leakage or accidental use of leaked data or capabilities.

`CReturn` has the simpler tasks of validating that any returned capability is global or `NULL`, clearing non-return registers, and popping and restoring `$pcc` and `$idc`:

```
/* 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();
```

The kernel accepts system calls only from classes that have the software-defined `User_Syscall` permission, ensuring that access to system resources can be mediated by in-process system objects.

7 Evaluation

7.1 Memory-protection performance

CHERI enables the compiler to represent all pointers as capabilities. This primarily imposes a cost in memory footprint due to larger pointers. To measure worst-case overheads for linked-list and tree-traversal operations, we compiled the pointer-intensive Olden microbenchmark suite to use capabilities for all pointers. The average execution-time overhead is 46%. This is near the overhead limit, with less pointer-heavy applications typically exhibiting an overhead well below 10%. Reducing the size of capabilities and making more selective use of capabilities present further opportunities for optimization. Detailed results are available in previous publications [19, 2].

7.2 Domain-crossing optimizations

To better understand object invocation costs with CHERI, we traced *best-case* invoke and return for a zero-byte `memcpy`. Figure 2 shows the cycle costs of the two domain transitions and sandboxed workload divided into phases.

A large cost lies in validating that capability arguments and return values conform to CheriBSD `CCall` semantics and capability-flow policies. Incorporating object type checking into the `CCall` instruction achieves modest gains of 44 cycles (5.5%). Checking for local capability arguments is costly, as a sequence of seven instruction is required for each of ten argument registers. An ISA extension could reduce this cost, but as the validation is specific to CheriBSD's compartment memory model, there is a tradeoff between generality and performance.

Clearing unused argument and return registers (including capabilities) to limit leaked data and attacks is also a significant cost. The `CClearRegs` instruction, with a modified kernel and compiler, experi-

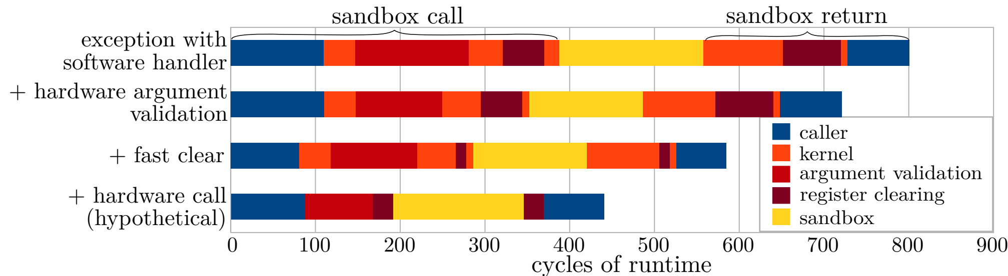


Figure 2: Total cycle count, spanning userspace and kernel, for a zero-byte `memcpy` in a sandbox

ences a further reduction of 172 cycles (21%).

Other significant costs include saving and restoring callee-save registers (12 general purpose, 11 capability), manipulating the trusted stack, trap overhead (four-fold for the call and return sequence), and cache and TLB usage if invocations are infrequent.

The final bar on Figure 2 shows a hypothetical pure-hardware, single-cycle domain crossing similar to the M-Machine [1]. This instruction unseals the code and data operands if their types match, and jumps directly into the sandbox. This hypothetical model sacrifices the trusted intermediary implemented by the kernel handler, losing the trusted stack and hence call-return semantics. It would lose the added assurance of having register clearing performed by an intermediary and system-wide guarantees as to the capability-flow policy. This model would eliminate another 145 cycles from the fastest software-handler case (18.1%), but at the expense of security and flexibility in the software model.

7.3 Domain-crossing performance

We performed an experiment to compare the cost of different compartmentalization mechanisms, and explore how they scale with the quantity of data transferred across protection boundaries:

pipe transfers to a sandbox process via a UNIX pipe

shmem transfers to a sandbox process through shared memory using a semaphore for synchronization

CHERI transfers to a in-process sandbox via a CHERI capability and `CCall`

func performs a simple unprotected function call to `memcpy`

Figure 3 shows the execution time for each method as the payload increases. While `memcpy` dominates as the payload grows, each domain-crossing mechanism has different fixed costs. A CHERI call/return gives a fixed overhead of around 500 cycles over a function call, whereas the other cases have much higher overhead (>17,000 cycles) from system calls and OS synchronization. The overhead for all shared-memory implementations is dominated by the cost of domain crossing for small payloads, but by the cost of cache and then TLB misses at larger sizes. The pipe case performs additional data copies and converges to about 6 times more expensive for larger data sizes.

7.4 Macrobenchmark: library compartmentalization

Transparent and efficient library compartmentalization is a major benefit of the CHERI approach. We demonstrate this through a compartmentalized `zlib` that performs compression in a sandbox, allowing all linked applications to receive security benefit. In Figure 4, we show the time `gzip` takes to compress files of varying sizes using CHERI compartmentalization, process-based Capsicum compartmentalization, and an unmodified `zlib` library. Process-based compartmentalization incurs linear overhead as it must transfer data using IPC, whereas CHERI shares memory using capabilities and experiences a small, near-constant, overhead due to domain-switch costs.

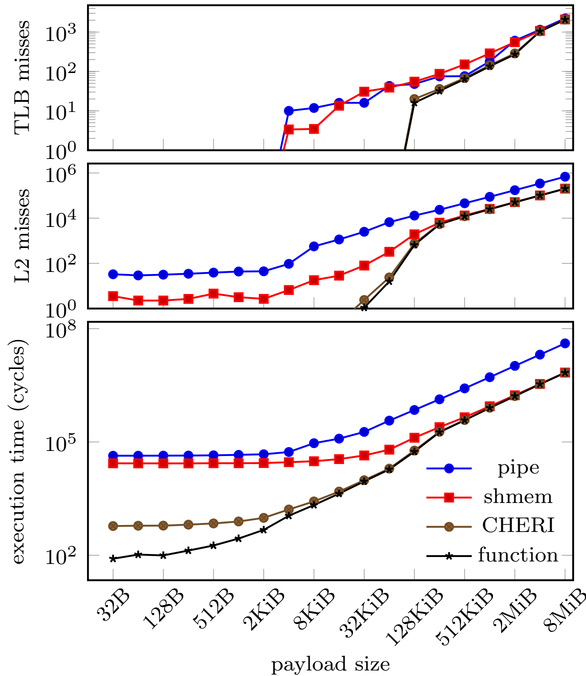


Figure 3: Comparison of domain crossing methods – absolute cycle cost (log-log)

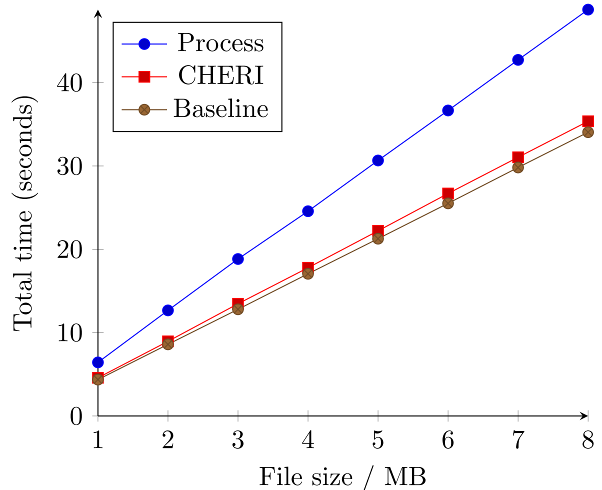


Figure 4: Compression time for `gzip` with library compartmentalization

8 Related work

Capability systems have a long history [3, 8, 13], with hardware-software systems such as the tagged and typed-object PSOS design [10] and the CAP [17] implementation – ideas adopted in operating systems such as HYDRA [20], and seL4 [6], and FreeBSD using Capsicum [15].

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. Whereas HYDRA used an MMU-based model with kernel-implemented capabilities, CHERI capabilities are represented in the ISA.

CHERI is also strongly influenced by M-Machine [1], which implemented fine-grained memory capabilities with tagged memory. 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 a conventional MMU, process model, C language, and interoperable ABIs.

Prior hardware research has explored other models for efficient domain crossing. Mondriaan is an access-control-centered approach based on a table-based mechanisms representing in-address-space security domains, and runs an adaptation of Linux [18]. PUMP provides a software-defined tagged model based on a clean-slate ISA approach, able to constrain information flow [4]. CODOMs provides code-centric, rather than object-centric, domains, where the current PC determines the accessible memory based on tags attached to page-table entries [14]. It provides low-latency switching by jumping between code domains but restricts each domain to a single instance. Data can be passed between domains using

fine-grained capabilities, but no fine-grained memory or pointer safety is provided within domains. Revocation is provided either by "synchronous" capabilities that survive in registers only for the duration of a cross-domain call, or using in-memory counters to invalidate "asynchronous" capabilities referencing the counter.

Further related work is considered in our papers on CHERI memory protection [19], C-language support [2], and compartmentalization [16].

9 Availability

We have released the CHERI hardware and software stacks, specifications, and manuals, as open source:

`http://www.cheri-cpu.org/`

Our experimental data is available at:

`https://www.cl.cam.ac.uk/research/security/ctsrd/data/`

10 Conclusion

Software compartmentalization is a critical and widely deployed vulnerability mitigation technique. However, compartmentalization scalability is limited by current processs architectures – in particular, the use of table-oriented virtual addressing as the means of constructing isolated compartments, and IPC as the means of communicating between them. We describe how CHERI memory protection can – with only minor additions and optimizations – supplement virtual addressing as a scheme to support fast domain switching with efficient shared memory. We demonstrate substantial performance improvement, and a high level of compatibility with current systems-software designs.

11 Acknowledgments

We thank Ross Anderson, Ruslan Bukin, Gregory Chadwick, Steve Hand, Wojciech Koszek, Bob Laddaga, Patrick Lincoln, Ilias Marinos, Andrew W. Moore, Alan Mujumdar, Prashanth Mundkur, Philip Paeps, Howie Shrobe, Stu Wagner, and Bjoern Zeeb, as well as our anonymous reviewers, 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 also acknowledge the EPSRC REMS Programme Grant [EP/K008528/1], the EPSRC Impact Acceleration Account [EP/K503757/1], EPSRC/ARM iCASE studentship [13220009], Microsoft studentship [MRS2011-031], the Isaac Newton Trust, the UK Higher Education Innovation Fund (HEIF), Thales E-Security, and Google, Inc.

References

[1] 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.

[2] 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.

[3] Dennis, J. B., and Van Horn, E. C. Programming semantics for multiprogrammed computations. *Communications of the ACM 9*, 3 (1966), 143–155.

[4] Dhawan, U., Hritcu, C., Rubin, R., Vasilakis, N., Chiricescu, S., Smith, J. M., Knight, T. F., Pierce, B. C., and DeHon, A. Architectural Support for Software-Defined Metadata Processing. In *20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (March 2015), ACM.

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

[6] 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. *Communications of the ACM 53* (June 2009), 107–115.

[7] 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.

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

[9] McKusick, M. K., Neville-Neil, G. V., and Watson, R. N. M. *The Design and Implementation of the FreeBSD Operating System*. Pearson, 2014.

[10] 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.

[11] Provos, N., Friedl, M., and Honeyman, P. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium* (2003), USENIX.

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

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

[14] Vilanova, L., Ben-Yehuda, M., Navarro, N., Etsion, Y., and Valero, M. CODOMs: Protecting software with code-centric memory domains. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Piscataway, NJ, USA, 2014), ISCA '14, IEEE Press, pp. 469–480.

[15] 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.

[16] Watson, R. N. M., Woodruff, J., Neumann, P. G., Moore, S. W., Anderson, J., Chisnall, D., Dave, N., s Davis, B., Gudka, K., Laurie, B., Murdoch, S. J., Norton, R., Roe, M., Son, S., and Vadera, M. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *Proceedings of the 36th IEEE Symposium on Security and Privacy* (May 2015).

[17] Wilkes, M., and Needham, R. *The Cambridge CAP computer and its operating system*. Elsevier North Holland, New York, 1979.

[18] Witchel, E., Cates, J., and Asanović, K. Mondrian memory protection. *ACM SIGPLAN Notices 37*, 10 (2002), 304–316.

[19] 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).

[20] Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., and Pollack, F. HYDRA: the kernel of a multiprocessor operating system. *Communications of the ACM 17*, 6 (1974), 337–345.