Balancing Disruption and Deployability in the CHERI Instruction-Set Architecture (ISA)

Robert N. M. Watson[†], Peter G. Neumann[*], and Simon W. Moore[†]
[†]University of Cambridge and [*]SRI International

12 February 2017

Introduction

For over two-and-a-half decades, dating to the first widespread commercial deployment of the Internet, commodity processor architectures have failed to provide robust and secure foundations for communication and commerce. This is in large part due to the omission of architectural features allowing efficient implementation of the *Principle of Least Privilege*, which dictates that software runs only with the rights it requires to operate [19, 20]. Without this support, the impact of inevitable vulnerabilities is multiplied as successful attackers gain easy access to unnecessary rights – and often, *all* rights – in software systems.

This omission is most visible at two levels of software abstraction: low-level code execution occurs with an excess of rights facilitating easy attacker manipulation, and higher-level encapsulation goals are poorly supported due to inefficiency. First, virtual addresses and C-language pointers (the references through which code and data are accessed) are implemented using unprotected and unconstrained integers, and are hence frequently exploited in attacks that escalate to arbitrary code execution. Second, compartmentalized software designs that constrain higher-level aspects of program behavior, mitigating lower-level vulnerabilities, scale poorly with current Memory Management Units (MMUs) – imposing a high penalty on use. Together, these gaps cause our most security-critical C-language software (e.g., operating systems, web browsers, and language runtimes) to offer asymmetric advantage to attackers in which the defender must make no mistakes, and the attacker can exploit a single mistake to gain total control. This is a dangerous status quo for contemporary network-connected ecosystems, whether mobile devices, embedded systems, or servers.

Supported by DARPA's CRASH research program, the CTSRD Project has sought to address this concern through a clean-slate re-design project to create the *Capability Hardware Enhanced RISC Instructions (CHERI)* Instruction-Set Architecture (ISA), processor prototype, and software stack. Our goal has been to address these two omissions from the ground up, providing strong architectural support for the principle of least privilege, offering new innate pro-

tections that naturally mitigate inevitable software bugs. We have drawn on over four decades of computer-security research dating to early systems and security projects [34, 15, 32, 8, 3, 10, 1], hardware-software co-design methodology, principled system design [16], and also recent insights into techniques for hybridizing capability-system approaches with OS and programming-language design [24, 12]. The surprising result has been a hardware-software approach that disrupts key tools used by attackers while continuing to support current software structures, and hence can be adopted within contemporary system designs.

Through CHERI, we seek to insert secure computer-architecture foundations beneath today's system software stacks with a minimum of disruption – while bringing fundamental improvements in robustness and security made efficient only through new hardware primitives. Key technical contributions include: the hybridization of a strong capability-system approach with a conventional MMU-based RISC design, permitting highly compatible integration with current OS and application designs; convergence of the C-language pointer semantics with capabilities; new programming models supporting fine-grained compartmentalization within conventional processes; and highly efficient architectural and microarchitectural approaches to memory protection. Each of these has been validated through full-stack hardware and software prototypes required to evaluate security, compatibility, and performance impact. In this chapter, we consider CHERI from four perspectives:

Methodology and philosophy of approach We describe the problem we seek to address, our motivating use cases, our key technical objectives, and our methodology and philosophy of approach grounded in hardware-software co-design. (Section 1)

CHERI architecture and software We present the key technical aspects of the work, including our goals of hybridizing a *capability-system model* with MMU-based operating systems (OSes) and the C programming language, and introduce our approach to fine-grained memory protection and scalable compartmentalization. (Section 2)

Research and development cycle We review the development of the key technical elements in CHERI, and the iterative cycle through six major instruction-set revisions over a (thus far) 7-year timeline. (Section 3)

Potential for impact We conclude by considering lessons learned, as well as the potential opportunities for impact within current system designs. We believe that these lessons apply broadly to other work on architectural security. We also consider next directions for the CTSRD project as we enter a further two years of research and development on CHERI. (Section 4)

1 Problem, Opportunity, Goals, and Approach

Despite half a century of research into computer systems and software design, it is clear that security remains a challenging problem – and an increasingly critical problem as computer-based technologies find ever expanding deployment in all aspects of contemporary life, from mobile communications devices to self-driving cars and medical equipment. There are many contributing factors to this problem, including the asymmetric advantage held by attackers over defenders (which cause minor engineering mistakes to lead to undue vulnerability), the difficulties in assessing – and comparing – the security of systems, and market pressures to deliver products sooner rather than in a well-engineered state. Perhaps most influential is the pressure for backward compatibility, required to allow current software stacks to run undisturbed on new generations of systems, as well as to move seamlessly across devices (and vendors), locking in least-common-denominator design choices, and preventing the deployment of more disruptive improvements that serve security.

Both the current state, and worse, the current direction, support a view that today's computer architectures (which underlie phenomenal growth of computer-based systems) are fundamentally "unfit for purpose": Rather than providing a firm foundation on which higher-level technologies can rest, they undermine attempts to build secure systems that depend on them. To address this problem, we require designs that mitigate, rather than emphasize, inevitable bugs, and offer strong and well-understood protections on which larger-scale systems can be built. Such technologies can be successful only if transparently adoptable by end users – and, ideally, also many software developers. On the other hand, the resulting improvement must be dramatic to justify adopting substantive architectural change, and while catering to short-term problems, must also offer a longer-term architectural vision able to support further benefit as greater investment is made.

1.1 Opportunity

Despite the challenge this problem represents, there are also reasons for hope:

- Improvements in physical fabrication technologies have allowed more complex computer architectures to be supported, while sustaining performance growth and reducing energy use. This creates the opportunity to invest greater computational resources in security at lower incremental cost.
- The desire to bring the benefits of electronic commerce to devices ranging from computer servers to phones and watches has created a strong financial incentive for computer vendors to improve security. This creates not just compliance obligations, but also the significant exposure to potential direct (and sometimes existential) financial loss for companies.
- There is increasing appetite for mitigation techniques on existing hardware from stack canaries and Address Space Layout Randomization (ASLR)

that are transparent to software but impact memory usage [21], through to process-based compartmentalization that is disruptive software [17, 7, 24, 18, 23]: function calls become Inter-Process Communication (IPC) and additional virtual address spaces impact MMU efficiency. These techniques increasingly impact on performance on current architectures, but, due to a reliance on randomization, also increase in-field non-determinism, which affects maintainability. Recovering lost performance, reducing complexity, and restoring software determinism are all potential benefits to better architectural protection.

- Recent modest changes in architecture, such as adopting the dual-ISA world of Intel x86 on the desktop and ARM on mobile devices (motivated by diverse energy and performance requirements), and similarly the transition from 32-bit to 64-bit, have acclimated software developers and product vendors to the need for minor disruption, maintaining multi-architecture software stacks. They have accepted and benefited from minor changes required to better abstract pointers (by reducing confusion with integers in order to span 32-bit and 64-bit ISAs), and supporting legacy environments (such as 32-bit compatibility 64-bit operating systems). Where further disruption can be aligned with these existing patterns, it may be similarly tolerated as an accepted and well-understood set of costs.

- Multiple decades of system design evolution have led to a strong consensus on how to integrate current architectural security features (such as MMUs) into software stacks, and similarly on software structures such as operating systems, programming languages/compilers, and applications. While that baseline omits many critical security functions, its existence means that new security technologies could be consistently applied (and incrementally composed) across multiple architectures and software structures.

- While security principles (such as the Principle of Least Privilege) have been known for decades, there is recent new understanding arising out of the security-research community about how to deploy those principles incrementally by hybridizing those approaches with current system and language designs. This creates the opportunity to consistently introduce disruptive new security features incrementally within current designs, as well as to deploy use of these principles at multiple levels of abstraction, offering strong mitigation potential against as-yet undiscovered classes of vulnerabilities and exploit techniques.

- Developments in formal methodology relating to automation and large-scale application of theorem-proving tools give us the confidence to approach more tightly integrated security designs – but also dramatically improve the efficiency of a small team working in the complex arena of hardware-software co-design.

1.2 Technical Objectives and Implementation

From a purely technical perspective, the aim of the CHERI project is to introduce architectural support for the principle of least privilege in order to encourage its direct utilization at all levels of the software stack. Current computer architectures make this extremely difficult as they impose substantial performance, robustness, compatibility, and complexity penalties in doing so – strongly disincentivizing adoption of such approaches in off-the-shelf system designs despite the potential to mitigate broad classes of known (and also as-yet unknown) vulnerability classes.

Low-level Trusted Computing Bases (TCBs) are typically written in memory-unsafe languages such as C and C++, which do not offer compatible or performant protection against pointer corruption, buffer overflows, or other vulnerabilities arising from that lack of safety not offered directly by the architecture. Similarly, software compartmentalization, which mitigates both low-level vulnerabilities grounded in program representation and high-level application vulnerabilities grounded in logical bugs, is poorly supported by current MMUs, leading to substantial (crippling) loss of programmability and performance as the technique is deployed.

CHERI also seeks to minimize disruption of current designs, in order to support incremental adoption with significant transparency: Ideally, CHERI could be "slid under" current software stacks (such as Apple's iOS ecosystem, or Google's Android ecosystem), allowing non-disruptive introduction, yet providing an immediate reward for adoption. This requires supporting current low-level languages such as C and C++ more safely, but also cleanly supplementing MMU-based programming models required to support current operating systems and virtualization techniques. These goals have directed many key design choices in the CHERI-MIPS ISA.

1.3 Hardware-Software Co-Design Methodology

Changes to the hardware-software interface are necessarily disruptive. The ISA is a "narrow waist" abstraction that allows hardware designers to pursue sophisticated optimization strategies (e.g., to exploit parallelism), while software developers can simultaneously depend on a (largely unchanging) interface to build successively larger and more complex artifacts. Stable ISAs have allowed the development of operating systems and application suites that can operate successfully on a range of systems, and that outlast the specific platforms on which they were developed. This structure is inherently predisposed to non-disruption, as platforms that incur lower adoption costs will be preferred to those that have higher costs. However, substantive changes in underlying program representation, such as to support greater memory safety or fine-grained compartmentalization required to dramatically improve security, require changes to the ISA. We therefore aimed to:

- Iteratively explore disruptions to the ISA, projecting changes both up into the software stack including operating systems, compilers, and appli-

cations (to assess impact on compatibility and security), as well as down into microarchitecture (assessing impact on performance and viability).

- Start with a conventional and well-established 64-bit RISC ISA, rather than re-invent the wheel for general-purpose computation, to benefit from existing mature software stacks that could then be used for validation.

- Employ realistic open-source software artifacts, including the FreeBSD operating system, Clang/LLVM compiler suite, and an open-source application corpus, to ensure that experiments were run with suitable scale, complexity, performance footprint, and idiomatic use.

- Employ realistic hardware artifacts, developing multiple FPGA soft-core based processor prototypes able to validate key questions about integration with components such as the pipeline and memory hierarchy, as well as support performance validation for the full stack including software.

- Employ formal models of the ISA, to provide an executable gold model for testing, from which tests can be automatically generated, and against which theorem proving can be deployed to ensure that key properties relied on for software security actually hold.

- Pursue the hypothesis that historic capability-system models, designed to support implementation of the principle of least privilege, can be hybridized with current software approaches to support compatible and efficient fine-grained memory protection and compartmentalization.

- Take an initially purist capability-system view, incrementally adapting that model towards one able to efficiently yet safely support the majority of current software use. This approach allowed us to retain well-understood monotonicity and encapsulation properties, as well as pursue capturing notions of explicit valid provenance enforcement and intentional use not well characterized in prior capability-system work. Appropriately but uncompromisingly represented, these properties have proven to align remarkably well with current OS and language designs.

- Aim specifically to cleanly compose with conventional MMUs and MMU-based software designs by providing an in-address-space protection model, as well as be able to represent C-language pointers as capabilities.

- Support incremental adoption, allowing significant benefit to be gained through modest efforts (such as re-compiling) for selected software, while not disrupting binary-compatible execution of legacy applications. Likewise, support incremental deployment of more disruptive compartmentalization into key software through greater (but selective) investment.

- Provide primitives that offer immediate short-term benefit (e.g., invulnerability to common pointer-based exploit techniques, scalable sandboxing

of libraries in key software packages), while also offering a longer-term vision for future software structure grounded in strong memory safety and fine-grained compartmentalization.

2 CHERI Architecture and Software

In this section, we briefly describe the CHERI-MIPS ISA and its use in protecting pointers in generated code, as well as software compartmentalization. Several software models can be layered over CHERI, including hybrid operating systems that employ the MMU for address-space separation, and CHERI for compiler-managed, capability-based in-address-space memory protection (see Figure 1). This description is roughly synchronized to CHERI ISAv6 as published in May 2017 [27]. While we have prototyped CHERI with respect to 64-bit MIPS, the approach described in this section implements a more general protection model potentially applicable to a range of ISAs including Intel x86, RISC-V, and ARM.

2.1 The CHERI-MIPS Instruction-Set Architecture (ISA)

In CHERI-MIPS, pointers may be represented as either integer virtual addresses or *tagged capabilities* that atomically combine virtual addresses with additional protection metadata. CHERI-MIPS supplements the general-purpose 64-bit MIPS register file with a *capability register file* that holds a set of 256-bit *capability registers* (see Figure 2). A later 128-bit in-memory representation employs bounds-compression techniques to reduce the memory overhead, trading off reduced bounds precision on large allocations against pointer size. *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 – for example, restricting use of a pointer for load, store, or execute. 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*.

Most capability registers are available to compiler and 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* (`$ddc`) interposes on (or blocks) conventional MIPS loads and stores. Two special capabilities are available to exception handlers: the *kernel code capability* (`$kcc`) and *kernel data capability* (`$kdc`).

Capability instructions employ *guarded manipulation* to implement *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

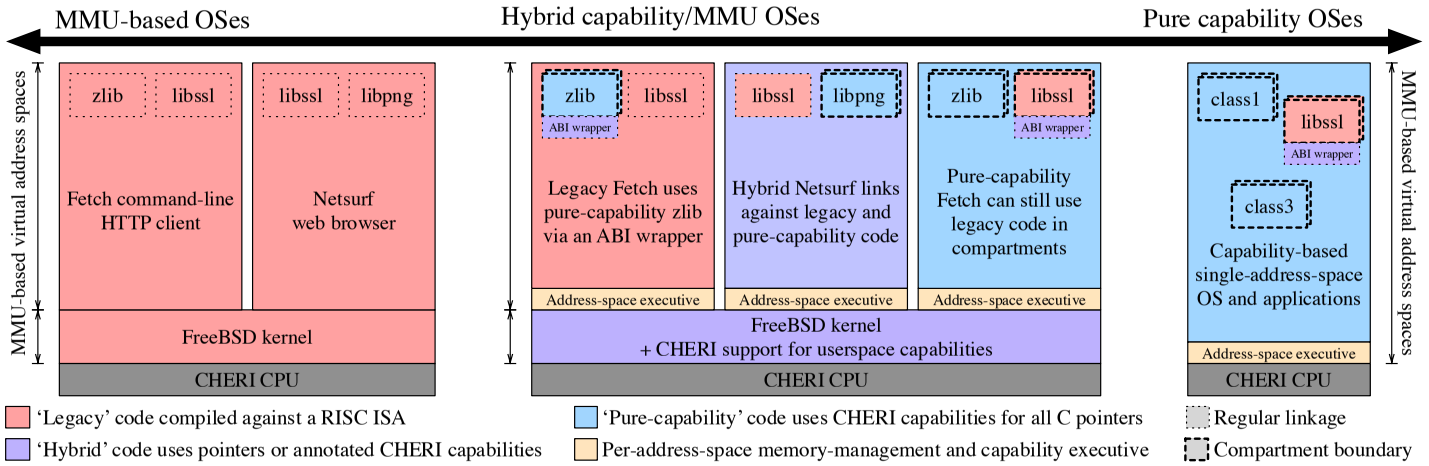

Figure 1: CHERI supports a spectrum of hardware-software architectures.

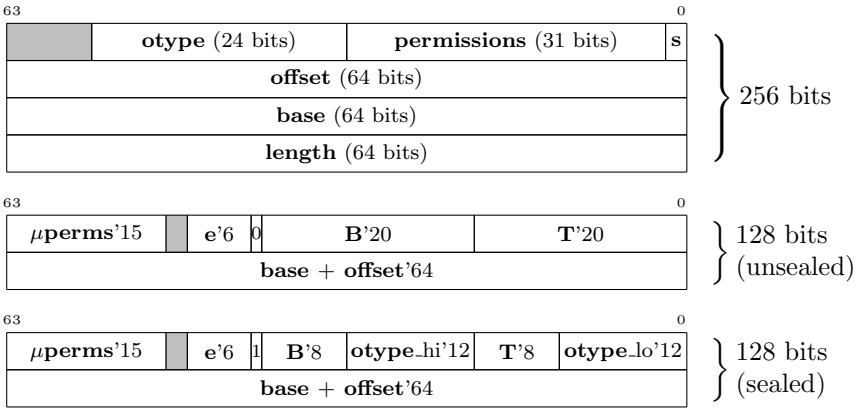

Figure 2: CHERI-256 and CHERI-128 memory representations for capabilities

loads from, capabilities in memory are atomic with their tags, allowing safe concurrent access from multiple cores. Tags enforce the *integrity* and valid *provenance* of a pointer by ensuring that only values derived from a valid pointer, via valid transformations, can be dereferenced. 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. Capability-based compartmentalization is provided by the encapsulation instructions that operate on *sealed* capabilities.

Several architectural features are added in order to support software compartmentalization (see Section 2.3). *Sealed capabilities* allow capabilities to be made immutable and non-dereferenceable, allowing them to support software-defined object implementations while retaining strong integrity and provenance properties. *Object types* in capabilities allow sets of capabilities to be linked in a non-forgeable manner, supporting more complex structures such as linked code and data capabilities implementing objects. A *hardware-accelerated object invocation* exception combines a set of fast-path checks with a software-defined exception handler to implement domain switching. *Fast register clearing instructions* allow the register file to be quickly cleared when transitioning domains, further improving domain-crossing performance.

The CHERI FPGA soft-core processor implements a capability register file, capability instructions, and tagged physical memory. Detailed descriptions of the prototype may be found in our published papers and technical repots [33, 2, 23, 25, 22].

2.2 Protecting Pointers with CHERI

Simply by recompiling C-code, all data pointers and code pointers are represented as capabilities. Despite the promiscuous use of pointers in C-code, the

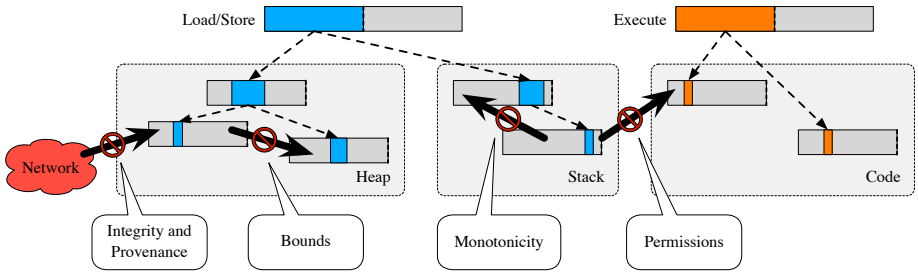

Figure 3: CHERI pointer provenance trees

vast majority of pointers have a provenance that is summarized as a tree in Figure 3. The following key properties emerge that allow important abstractions to be preserved:

Integrity and provenance of the capability are guaranteed by the validity tag, cleanly separating pointers from data. Attackers can no longer inject pointers via the network, as data writes will be tag-free, preventing later dereference. The compiler represents all return addresses as capabilities, thereby making return-oriented programming (ROP) attacks much harder because the attacker not only has to overwrite the return address, but also has to ensure it is a code capability with integrity and provenance.

Bounds (and the tag) prevent a capability referring to one object being used to access another. Bounds prevent buffer overflow and over-read attacks, for example preventing bugs such as Heartbleed.

Monotonicity guarantees that bounds and permissions can never be increased, preventing privilege escalation.

Permissions prevent a number of attacks including code modification, or in the case of a JIT compiler, providing fine-grained control over what can generate code and where it can place that code.

Capabilities also allow the *Principle of Intentional Use* to be expressed: where multiple rights are available to a program, the selection of rights used to authorize work on behalf of the program is explicit [29]. The effect of preserving this principle during the compilation process is to avoid the accidental or unintended exercise of rights that could lead to a violation of the intended policy. For example, memory loads and stores are with respect to an explicitly named capability register, and instruction fetches are via the program-counter capability, rather the register used for load, store, or fetch being selected implicitly from a table via an associative lookup. The effect of this principle is to counter what are classically known as 'confused deputy' problems, in which a program will unintentionally exercise a privilege that it holds legitimately, but on behalf of another party who does not (and should not) hold that privilege. This principle, common to many capability systems, has been applied throughout the CHERI

design, from architectural privilege management (e.g., operations via explicit capability registers) through to privilege management by software abstractions such as the CheriBSD object-capability systems, which are enabled in this by sealed capabilities.

2.3 Software Compartmentalization

Software compartmentalization is a fundamental abstraction that limits privileges and further attack surfaces available to attackers [6, 17, 24]. In compartmentalization, applications are decomposed into isolated ("sandboxed") components that are granted only selected access to system and application resources. 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.

Unlike more specific exploit mitigation techniques (which targets attack-vector characteristics such as remote code injection), compartmentalization does not depend on knowledge of specific attack vectors, and is resistant to an arms race as attack and defence co-evolve. Fine-grained compartmentalization improves mitigation by virtue of the principle 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.

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 [17] and Chromium [18].

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 [15, 34, 8, 24]. *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.

We have used CHERI's ISA facilities as a foundation to build a software *object-capability model* supporting orders of magnitude greater compartmentalization performance, and hence appropriate granularity, than current designs. We use sealed capabilities to build a *hardware-software domain-transition mechanism* and *programming model* suitable for safe communication between mutu-

ally distrusting software.

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).

In contrast to MMU-based approaches, CHERI-based compartmentalization optimizes sharing by allowing cheap delegation and avoiding aliasing problems experienced by TLBs as memory sharing increases [30]. This allows domain crossing to be performed at a low constant cost regardless of the amount of data sharing. These properties are critical to scaling up intra-application compartmentalization that is characterized by frequent domain crossings and extensive memory sharing. CHERI also eases programming for compartmentalized software by virtue of restoring a single address-space model, where MMUs imposed a multi-address-space model that programmers find difficult to reason about.

In addition to developing a high-performance compartmentalization mode, we have also explored how software static analysis can assist programmers in reasoning about decomposing software in order to accomplish mitigation objectives [4].

3 Research and Development

Between 2010 and 2017, six major versions of the CHERI-MIPS ISA developed a mature hybridization of conventional RISC architecture with a strong (but software-compatible) capability-system model. Key research and development milestones can be found in Figure 4 including major publications. The major ISA versions, with their development focuses, are described in Table 3. This work occurred in several major overlapping phases as aspects of the approach were proposed, refined, and stabilized through a blend of ISA design, integrated hardware and software prototyping, and validation of the combined stack.

2010-2015: Composing the MMU with a capability-system model

A key early design choice was that the capability-system model would be largely orthogonal to the current MMU-based virtual-memory model, yet also compose with it cleanly [33]. We chose to place the capability-system model "before" the MMU, causing capabilities to be interpreted with respect to the virtual, rather than physical, address space. This reflected the goal of providing fine-grained memory protection and compartmentalization within address spaces – i.e., with respect to the application-programmer model of memory.

Capabilities therefore protect and implement virtual addresses dereferenced in much the same way that integer pointers are interpreted in conventional

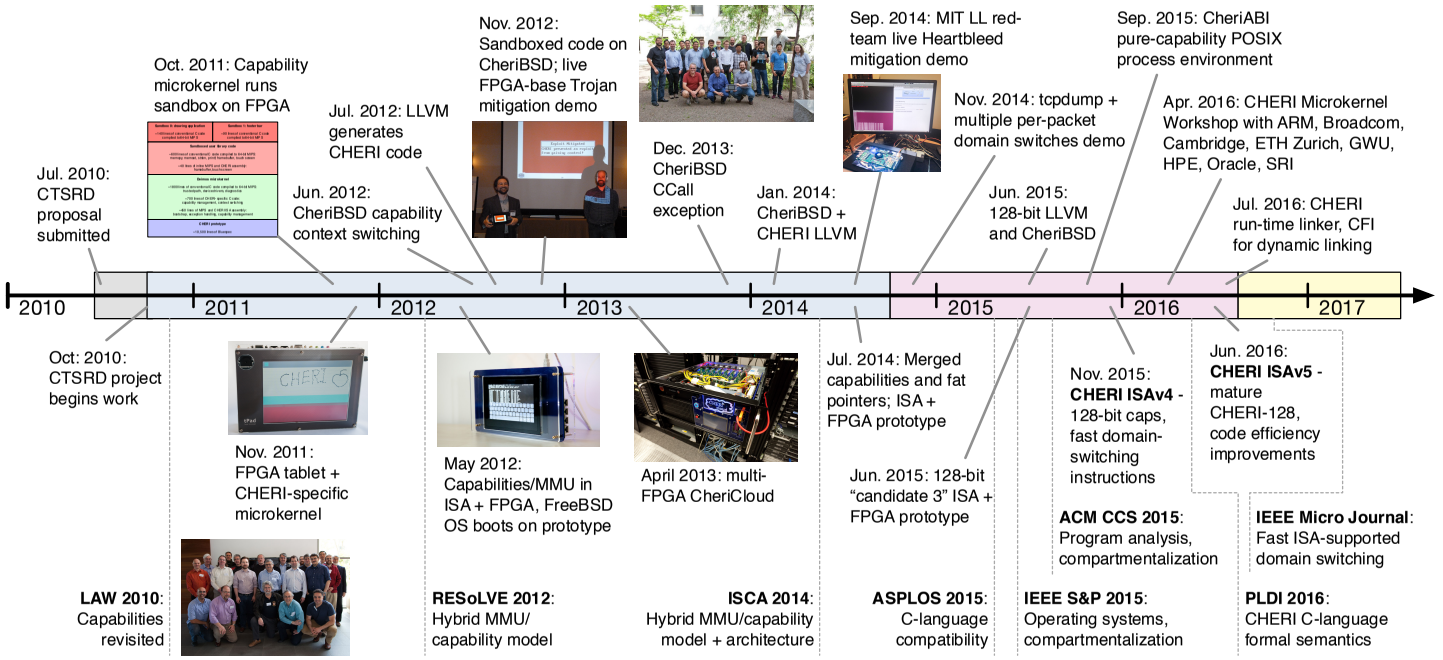

Figure 4: CHERI research and development timeline, 2010-2017

Table 1: CHERI ISA revisions and major development phases

Year(s)	Version	Description
2010-2012	ISAv1	RISC capability-system model w/64-bit MIPS Capability registers and tagged memory Guarded manipulation of registers
2012	ISAv2	Extended tagging to capability registers Capability-aware exception handling MMU-based OS with CHERI support
2014	ISAv3 [26]	Fat pointers + capabilities, compiler Instructions to optimize hybrid code Sealed capabilities, `CCall`/`CReturn`
2015	ISAv4 [28]	MMU-CHERI integration (TLB permissions) ISA support for compressed capabilities Hardware-accelerated domain switching Multicore instructions: `LL`/`SC` variants
2016	ISAv5 [29]	CHERI-128 compressed capability model Improved generated code efficiency Initial in-kernel privilege limitations
2017	ISAv6 [27]	Mature kernel privilege limitations Further generated code efficiency CHERI-x86 and CHERI-RISC-V sketches Jump-based protection-domain transition

architectures. Exceptions allow controlled escape from the capability model by providing access to privileged capability registers, and execution in privileged rings grants the ability to manipulate the virtual address space, controlling the interpretation of virtual addresses embedded in capabilities.

This approach tightly integrates the capability-system model with the pipeline and register file, requiring that capabilities be first-class primitives managed by the compiler, held in registers, and so on. In order to protect capabilities in the virtual address space, we chose to physically tag them, distinguishing strongly protected pointers from ordinary data, in turn extending the implementation of physical memory, but also making that protection entirely independent from (and non-bypassable by) the MMU mechanism.

2012-2014: Composing C pointers with the capability-system mode

Another key early design choice was the goal of using capabilities to implement C-language pointers – initially discretionarily (i.e., as annotated in the language), and later ubiquitously (i.e., for all virtual addresses in a more-secure program). This required an inevitable negotiation between C-language semantics and the capability-system model, in order to ensure strong compatibility with current software [2, 11].

For example, C embeds a strong notion that pointers point within buffers. This requires that CHERI capabilities distinguish the notion of current virtual address from the bounds of the containing buffer – while also still providing strong integrity protection to the virtual address. This led us to compose fat-pointer [5, 13, 14] and capability semantics as the capability-system model evolved.

Similarly, we wished to allow all pointers to be represented as capabilities – including those embedded within other data structures – leading naturally to a choice to mandatorily tag pointers in memory. A less obvious implication of this approach is that operations such as memory copying must be capability-oblivious, maintaining the tag across pointer-propagating memory operations, requiring that data and capabilities not only be intermingled in memory, but also in register representation. Capability registers are therefore also tagged, allowing them to hold data or capabilities, preserving provenance transparently.

As part of this work, we also assisted with the development of new formal semantics for the C programming language, ensuring that we met the practical requirements of C programs, but also assisting in formalizing the protection properties we offer (e.g., strong protection of provenance validity grounded in an implied pointer provenance model in C).

CHERI should be viewed as providing primitives to support strong C-language pointer protection, rather than as directly implementing that protection: it is the responsibility of the compiler (and also operating system and runtime) to employ capabilities to enforce protections where desired – whether by specific memory type, based on language annotations, or more universally. The compiler can also perform analyses to trade off source-code and binary compatibility, enforcing protection opportunistically in responding to various potential policies on tolerance to disruption.

2014-2015: Fine-grained compartmentalization

A key goal of our approach was to differentiate virtualization (requiring table-based lookups, and already implemented by the MMU) from protection (now implemented as a constant-time extension to the pointer primitive), which would avoid table-oriented overheads being imposed on protection. This applies to C-language protection, but also to the implementation of higher-level security constructs such as compartmentalization [31, 30].

Compartmentalization depends on two underlying elements: strong isolation and controlled communication bridging that isolation. Underlying monotonicity in capabilities – i.e., that a delegated reference to a set of rights cannot be broadened to include additional rights – directly supports the construction of confined components within address spaces. Using this approach, we can place code in execution with only limited access to virtual memory, constructing "sandboxes" (and other more complex structures) within conventional processes. The CHERI exception model permits transition to a more privileged component – e.g., the operating-system kernel or language runtime – allowing the second foundation, controlled communication, to be implemented.

Compartmentalization is facilitated by further extensions to the capability

model, including a notion of "sealed" (or encapsulated capabilities). In CHERI, this is implemented as a software-defined capability: one that has no hardware interpretation (i.e., cannot be dereferenced), and also strong encapsulation (i.e., whose fields are immutable). Other aspects of the model include a type mechanism allowing sealed code and data capabilities to be inextricably linked; pairs of sealed code capabilities and data capabilities can then be used to efficiently describe protection domains via an object-capability model. We provide some hardware assistance for protection-domain switching, providing straightforward parallel implementation of key checks, but leave the implementation of higher-level aspects of switching to the software implementation.

Here, as with C-language integration, it is critical that CHERI provide a general-purpose mechanism rather than enforce a specific policy: the sealed capability primitive can be used in a broad variety of ways to implement various compartmentalization models with a range of implied communication and event models for software. We have experimented with several such models, including a protection-domain crossing primitive modeled on a simple (but now strongly protected) function call, and also on asynchronous message passing. Our key performance goal was fixed (low) overhead similar to a function call, avoiding overheads that scale with quantity of memory shared (e.g., as is the case with table-oriented memory sharing configured using the MMU).

2015-2017: Architectural and microarchitectural efficiency

Side-by-side with development of a mature capability-based architectural model, we also explored the implications on performance. This led to iterative refinement of the ISA to improve generated code, but also substantive efforts to ensure that there was an efficient in-memory representation of capabilities, as well as microarchitectural implementations of key instructions.

A key goal was to maintain the principle of a load-store architecture by avoiding combining computations with memory accesses – already embodied by both historic and contemporary RISC architectures. While pointers are no longer conflated with integer values, a natural composition of the capability model and ISA maintains that structural goal without difficulty.

One important effort lay in the reduction from a 256-bit capability (capturing the requirements of software for 64-bit pointer, 64-bit upper bound, and 64-bit lower bound, as well as additional metadata such as permissions) to a 128-bit compressed representation. We took substantial inspiration from published work in pointer compression [9], but found that our C-language compatibility requirements imposed a quite different underlying model and representation. For example, it is strictly necessary to support the common C-language idiom of permitting out-of-bounds pointers (but not dereference), which had been precluded by many proposed schemes [2, 11]. Similarly, the need to support sealed capabilities led to efforts to characterize the tradeoff between the type space (the number of unique classes that can be in execution in a CHERI address space) and bounds precision (the alignment requirements imposed on sealed references).

Another significant effort lay in providing in-memory tags, which are not

directly supported by current DRAM layouts. In our initial implementation, we relied on a flat tag table (supported by a dedicated tag cache). This imposed a uniform (and quite high) overhead in additional DRAM accesses across all memory of roughly 10%. We have developed new microarchitectural techniques to improve emulated tag performance, based on a hierarchical table exploiting sparse use of pointers in memory, to reduce this overhead to $< 2\%$ even with very high pointer density (e.g., in language runtimes).

2016-2017: Kernel Compartmentalization

Our initial design focus was on supporting fine-grained memory protection within the user virtual address space, and implicitly, also compartmentalization. Beyond an initial microkernel brought up to validate early capability model variants, kernel prototypes through much of our project have eschewed use of capability-aware code in the kernel due to limitations of the compiler, but also because of a focus on large userspace TCBs such as compression libraries, language runtimes, web browsers, and so on, which are key attack surfaces.

We have more recently returned to in-kernel memory protection and compartmentalization, where the CHERI model in general carries through without change – code executing in the kernel is not fundamentally different from code executing in userspace. The key exception is a set of management instructions available to the kernel, able to manipulate the MMU (and hence the interpretation of capabilities), as well as control features such as interrupt delivery and exception handling. We are now extending CHERI to allow the capability mechanism to control access to these features so that code can be compartmentalized within the kernel. We are also pursuing changes to the exception-based domain-transition mechanism used in earlier ISA revisions that shift towards a jump-based model, which will avoid exception-related overheads in the microarchitecture.

3.1 CHERI ISAv6: Looking Beyond MIPS

As we wrap up work on CHERI ISAv6, we are looking beyond the 64-bit MIPS ISA on which we based our hardware-software co-design effort towards further ISAs. These range from the still-developing open-source RISC-V ISA (which strongly resembles the MIPS ISA and hence to which most CHERI ideas will apply with minor translation) to the widely used Intel x86-64 instruction set (which is quite far from the RISC foundations in which we have developed CHERI). This exploration has allowed us to derive a more general CHERI protection model from our work, rather than seeing CHERI as simply an extension to MIPS. We have focused on developing portable software-facing primitives and abstractions potentially supported by a variety of architectural expressions. We take some inspiration from the diverse range of MMU semantics and interfaces providing a common virtual-memory abstraction, and process model, across a broad range of architectures. New versions of the ISA specification also explore in much greater detail how architecture protection can be exploited by operating systems and compilers to reinforce program structure and mitigate

vulnerabilities.

4 Conclusion

Over the last seven years, the CTSRD project has performed intensive and iterative hardware-software co-design to develop the CHERI-MIPS ISA, focusing on introducing architectural support for the principle of least privilege. The resulting approach – a hybridization of architectural and software techniques building on capability systems, C-language memory safety, virtual memory, and operating systems – is surprisingly adoptable in large real-world software stacks. As described in Section 2.2, many security benefits can be achieved simply by recompiling current C-language TCBs with little or no source-code-level change, achieving fine-grained referential integrity and protection that mitigates many known classes of pointer-related exploit. With further investment in refactoring software described in Section 2.3, scalable support for fine-grained software compartmentalization opens the door to vulnerability and exploit-class non-specific mitigation, both accelerating current software compartmentalization, and supporting the introduction of much great compartmentalization.

By starting with a conventional RISC architecture and a C-language operating-system and application corpus, we have been able to demonstrate and validate our approach against large extant software stacks (e.g., the FreeBSD operating system), as well as provide an easier path to potential transition. Using FPGA-based prototypes, which allow a far tighter design cycle between hardware and software, we have also been able to support detailed resource and performance analyses, validating microarchitectural aspects of the approach. This hardware-software co-design approach has paid enormous dividends in forcing a vital iterative design and refinement process over several years. It is increasingly clear that the CHERI protection model is applicable to a broad range of architectures and microarchitectures, rather than being specific to the 64-bit MIPS architecture on which we have prototyped.

As the project enters its next two years (now seven years into a 4-year project!), we continue our focus on building larger demonstrations of the approach, maturing our software stack – including demonstrating how CHERI converges with OS design choices and the compiler stack, as well as improving performance through research into architectural and microarchitectural features such as capability compression and efficient hierarchical tag tables. We are also turning our attention from formal modeling (which has allowed us to precisely specify behaviors of the ISA for the purposes of informal reasoning and automated testing) to formal reasoning – yielding early proofs of key underlying security properties in the ISA, such as strong capability monotonicity, capability unforgeability, and protection-domain isolation. More information about the CHERI architecture and our ongoing work, along with open-source hardware and software artifacts, may be found on the CTSRD project website:

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

5 Acknowledgments

The work described in this chapter would not have been possible without numerous contributors to the CTSRD project over an extended period. This includes our co-authors on our published papers and technical reports: Jonathan Anderson, Ross Anderson, David Brazdil, Ruslan Bukin, David Chisnall, Brooks Davis, Nirav Dave, Khilan Gudka, Alexandre Joannou, Wojciech Koszek, Ben Laurie, James Lingard, A. Theodore Markettos, Ilias Marinos, J. Edward Maste, Justus Matthiesen, Kayvan Memarian, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton-Wright, Philip Paeps, Alex Richardson, Michael Roe, Colin Rothwell, Hassen Saidi, Peter Sewell, Stacey D. Son, Munraj Vadera, Jonathan Woodruff, and Hongyan Xia.

We also gratefully acknowledge the contributions of our colleagues at SRI International and the University of Cambridge, including John Baldwin, Hadrien Barral, Gregory Chadwick, Lawrence Esswood, Steven Hand, Stephen Kell, Patrick Lincoln, Anil Madhavapeddy, Alfredo Mazzinghi, Andrew W. Moore, Will Moreland, Alan Mujumdar, Prashanth Mundkur, Jeunese Payne, John Rushby, Hans Petter Selasky, Philip Withnall, and Bjoern Zeeb. Finally, the larger DARPA CRASH and MRC program communities, in which the CHERI project took place, provided critical support for our work; we are indebted to Howie Shrobe, Bob Laddaga, John Launchbury, Stu Wagner, Dan Adams, and Laurisa Goergen for their support throughout.

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], Isaac Newton Trust, 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] Gudka, K., Watson, R. N. M., Anderson, J., Chisnall, D., Davis, B., Laurie, B., Marinos, I., Neumann, P. G., and Richardson, A. Clean Application Compartmentalization with SOAAP. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS 2015)* (October 2015).

[5] 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* (Berkeley, CA, USA, 2002), USENIX.

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

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

[8] 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.

[9] 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 Conference on Computer and Communications Security* (November 2013), ACM.

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

[11] Memarian, K., Matthiesen, J., Lingard, J., Nienhuis, K., Chisnall, D., Watson, R. N., and Sewell, P. Into the depths of C: elaborating the de facto standards. In *Proceedings of PLDI 2016* (June 2016).

[12] 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).

[13] 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.

[14] Necula, G. C., McPeak, S., and Weimer, W. CCured: Type-safe retrofitting of legacy code. *ACM SIGPLAN Notices 37*, 1 (2002), 128–139.

[15] 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.

[16] Neumann, P. G. Principled assuredly trustworthy composable architectures. Tech. rep., Computer Science Laboratory, SRI International, Menlo Park, California, December 2004. http://www.csl.sri.com/neumann/chats4.html, .pdf, and .ps.

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

[18] 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.

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

[20] Saltzer, J., and Schroeder, M. The protection of information in computer systems. *Proceedings of the IEEE 63*, 9 (September 1975), 1278–1308.

[21] Szekeres, L., Payer, M., Wei, T., and Song, D. Eternal war in memory. In *IEEE Symposium on Security and Privacy* (2013).

[22] 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.

[23] Watson, R. N. M. A decade of OS access-control extensibility. *Communications of the ACM 56*, 2 (Feb. 2013).

[24] 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.

[25] Watson, R. N. M., Chisnall, D., Davis, B., Koszek, W., Moore, S. W., Murdoch, S. J., Neumann, P. G., and Woodruff, J. Capability Hardware Enhanced RISC Instructions: CHERI Programmer's Guide. Tech. Rep. UCAM-CL-TR-877, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, Nov. 2015.

[26] 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.

[27] Watson, R. N. M., Neumann, P. G., Woodruff, J., Roe, M., Anderson, J., Baldwin, J., Chisnall, D., Davis, B., Joannou, A., Laurie, B., Moore, S. W., Murdoch, S. J., Norton, R., Son, S., and Xia, H. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 6). Tech. Rep. UCAM-CL-TR-907, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500, Apr. 2017.

[28] Watson, R. N. M., Neumann, P. G., Woodruff, J., Roe, M., Anderson, J., Chisnall, D., Davis, B., Joannou, A., Laurie, B., Moore, S. W., Murdoch, S. J., Norton, R., and Son, S. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture. Tech. Rep. UCAM-CL-TR-876, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, Nov. 2015.

[29] Watson, R. N. M., Neumann, P. G., Woodruff, J., Roe, M., Anderson, J., Chisnall, D., Davis, B., Joannou, A., Laurie, B., Moore, S. W., Murdoch, S. J., Norton, R., Son, S., and Xia, H. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 5). Tech. Rep. UCAM-CL-TR-891, University of Cambridge, Computer Laboratory, June 2016.

[30] Watson, R. N. M., Norton, R. M., Woodruff, J., Moore, S. W., Neumann, P. G., Anderson, J., Chisnall, D., Davis, B., Laurie, B., Roe, M., Dave, N. H., Gudka, K., Joannou, A., Markettos, A. T., Maste, E., Murdoch, S. J., Rothwell, C., Son, S. D., and Vadera, M. Fast protection-domain crossing in the cheri capability-system architecture. *IEEE Micro 36*, 5 (Sept 2016), 38–49.

[31] 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).

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

[33] 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).

[34] 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.