Unpublished workshop paper for RESoLVE'12 - March 3, 2012 - London, UK

CHERI: a research platform deconflating hardware virtualization and protection

Robert N.M. Watson
University of Cambridge
robert.watson@cl.cam.ac.uk

Peter G. Neumann
SRI International
neumann@csl.sri.com

Jonathan Woodruff
University of Cambridge
jonathan.woodruff@cl.cam.ac.uk

Jonathan Anderson
University of Cambridge
jonathan.anderson@cl.cam.ac.uk

Ross Anderson
University of Cambridge
ross.anderson@cl.cam.ac.uk

Nirav Dave
SRI International
ndave@csl.sri.com

Ben Laurie
Google UK Ltd
benl@google.com

Simon W. Moore
University of Cambridge
simon.moore@cl.cam.ac.uk

Steven J. Murdoch
University of Cambridge
steven.murdoch@cl.cam.ac.uk

Philip Paeps
NixSys BVB
philip.paeps@cl.cam.ac.uk

Michael Roe
University of Cambridge
michael.roe@cl.cam.ac.uk

Hassen Saidi
SRI International
saidi@csl.sri.com

Abstract

Contemporary CPU architectures conflate virtualization and protection, imposing virtualization-related performance, programmability, and debuggability penalties on software requiring fine-grained protection. First observed in micro-kernel research, these problems are increasingly apparent in recent attempts to mitigate software vulnerabilities through application compartmentalisation. Capability Hardware Enhanced RISC Instructions (CHERI) extend RISC ISAs to support greater software compartmentalisation. CHERI's *hybrid capability model* provides fine-grained compartmentalisation within address spaces while maintaining software backward compatibility, which will allow the incremental deployment of fine-grained compartmentalisation in both our most trusted and least trustworthy C-language software stacks. We have implemented a 64-bit MIPS research soft core, BERI, as well as a capability coprocessor, and begun adapting commodity software packages (FreeBSD and Chromium) to execute on the platform.

1. Introduction

This workshop paper describes Capability Hardware Enhanced RISC Instructions (CHERI), an extension to the commodity 64-bit MIPS[1] instruction set architecture (ISA) to efficiently implement memory capabilities and the object-capability security model [14]. Using CHERI, we hope to provide enhanced hardware support for software compartmentalisation—the granular decomposition of software into isolated components in order to improve robustness, enforce security policies, and mitigate security vulnerabilities. Compartmentalisation combines abstraction, modularity, encapsulation, privilege, and separation, whose effects are not generally directly exposed to contemporary hardware.

[1] We hope also to investigate this approach with the new ARMv8 ISA.

Adoption of software compartmentalisation has historically been resisted due to its impacts on performance, programmability, and debuggability. Many of these problems result from the use of virtual memory techniques, which were developed to solve quite different problems in coarse-grained memory management and paging rather than to provide fine-grained protection. Virtual addressing also causes programmability and debugging issues in this context, as programmers are unnecessarily exposed to distributed system programming problems even in local application development. Despite these limitations, hostile computing environments have motivated increasing deployment of compartmentalisation in operating systems and applications [15, 16]. However, such efforts are plagued by problems of inefficiency, lack of scalability, and difficulty in programming and debugging—as we experienced first-hand during our work on Capsicum [22].

CHERI offers an alternative path by introducing new hardware protection features that are divorced from virtual memory features. CHERI systems retain an MMU, which can still be used for full-system and process virtualization, as well as paging and swapping. However, CHERI also offers a non-hierarchical, in-address-space protection model, grounded in capabilities, that can be managed without recourse to privileged CPU instructions. The CHERI capability model allows efficient implementation of in-address-space software compartmentalisation by providing fine-grained memory protection and trustworthy security domain transitions [6, 24]. The model is a *hybrid capability model*, which allows capability-enhanced software components to be tightly integrated with conventional software designs. The hybrid capability philosophy, first developed in the Capsicum operating system (but constrained by conventional CPU architectures), provides an incremental software adoption path, allowing capability system approaches to be deployed in our most trusted TCBs (OS kernels, system libraries, language run-times), as well as our least trustworthy software (im-

age decompression and video CODECs), all invisibly to other code in the system [22]. For example, CHERI is able to transparently execute capability-hardened libraries within unmodified applications, as well as the converse: sandboxing legacy software libraries within capability-aware software.

This workshop paper presents our early work in order to seek feedback from the research community, and so adopts a didactic style investigating our motivations, methodology, and future plans. We begin by considering in greater detail the historic conflation of virtualization with protection in hardware, and how this interacts with protection goals in software. We then describe our research approach, which is concerned with whether protection features are better placed in hardware or software. We explore our current prototype implementation, including development of a generalised platform for research into the hardware-software interface and its specific adaptation to security research, as well as a demonstration of a hybrid operating system combining capability-aware and legacy code. Finally, we consider possible techniques for evaluating this research, future directions, and related work.

2. Background

Extensive research in operating systems, virtual machines, and programming languages over the last four decades reflects an increasing desire for fine-grained, high-performance protection in software, ranging from the Berkeley Packet Filter (BPF) to the Java programming language [7, 11]. Motivations for granular decomposition of software include the desire to improve robustness against software bugs, mitigation of software vulnerabilities, and implementation of fine-grained security policies—approaches whose success derives from the principle of least privilege [18]. Experience from 1990s micro-kernel research (e.g., Mach), and more recently in application software compartmentalisation, suggests that current hardware-based protection techniques impose significant obstacles to successful mapping of software compartmentalisation into virtual-memory-based process models [1, 3, 15, 22]. This has led to increased interest in software-based static and dynamic enforcement techniques, such as SFI and Google NativeClient, which side-step hardware primitives, supporting continuous enforcement without recourse to CPU memory management units (MMUs) [21, 27]. However, these techniques efficiently address only hierarchical protection models: untrusted domains make fast "system calls" to trusted domains, rather than non-hierarchical models, such as communication between mutually untrusting software components within the same application. Recent attempts to employ software-based protection in non-hierarchical TCB environments (such as LXFI [10]) have seen substantial (even prohibitive) overhead.

Our work is motivated by our own experiences with software compartmentalisation in Capsicum, where we experienced first-hand the limitations of system software compartmentalisation:

- OS-provided process isolation scales poorly in the presence of tightly coupled (but mutually suspicious) application components. Shared memory leads to poor MMU behavior due to translation look-aside buffer (TLB) entry aliasing and high TLB miss rates. CPU vendors have optimised for hierarchical security domain transition (i.e., system call performance) rather than non-hierarchical relationships (inter-process communication).
- Programmability suffers as a result of unnecessarily virtualizing software structure when using processes. Multi-process programming leads quickly to distributed systems bugs due to error-prone IPC and data replication issues. Multi-process designs also limit debuggability, as current debugging tools operate poorly on distributed programs.

While virtual-machine- and language-derived protection models (such as those in Java) have proven effective for some purposes, the continuing lack of scalable protection techniques for low-level, C-language software trusted computing bases (TCBs)—such as operating system kernels and language run-times—is concerning. In single-user computer systems, even web browsers and office suites become part of the TCB, and have resisted migration to safer languages. Performance concerns are often cited as the primary objections to migrating from C and C++, but multi-million LoC legacy code bases are also an important factor.

We believe that the need for fine-grained compartmentalisation is clear: lack of robustness and resilience to attacks causes our most trusted software components (operating system kernels, language run-times, web browsers) to be seriously untrustworthy. Providing fine-grained compartmentalisation technology that is immediately accessible and incrementally deployable—i.e., compatible with current legacy C-language code bases—in the system software space is critical, even if we believe that type-safe, managed languages and formal methods will provide a longer-term solution.

3. Approach

Capability Hardware Enhanced RISC Instructions (CHERI) extend conventional RISC instruction set architectures (ISAs) with fine-grained protection independent of virtualization—i.e., from memory management units (MMUs). We draw on over forty years of architecture and security research into capability systems, starting with Multics [17] and CAP [24], and including PSOS [13], Hydra [26], Mach [1], Java [7], Joe-E [12], Eros [19], seL4 [8], and Capsicum[2] [22].

CHERI adopts Capsicum's *hybrid capability system* approach in order to blend conventional ISA compatibility with capability system functionality. This provides an incremental adoption path, allowing capability-oblivious conventional code to run both *around* and *within* capability-aware code. CHERI is deeply influenced by ideas from the programming language community, with obvious parallels between notions of hardware-supported typed objects and programming language-level versions of the same. The key design hypotheses in our work are:

1. Deconflating virtualization and protection will improve protection scalability, programmability, and debuggability.
2. Hardware protection features can directly support compartmentalisation of low-level TCBs, such as hypervisors, operating system kernels, and language run-times, to improve resistance to attacks—just as type-safe languages and higher-level language-based security features support high-level applications.
3. A hybrid capability model can capture both desired non-hierarchical security relationships and compatibility requirements to facilitate deployment with large existing code bases.
4. Retaining virtualization features in the presence of a capability model improves software compatibility, and also brings the recently explored benefits of full-system virtualisation to capability-based systems.
5. The judicious application of formal methods in hardware and low-level software design can provide a firm foundation for current and future systems.
6. Scientific exploration of protection features requires an experimental platform spanning hardware and software.

[2] Levy provides an excellent survey of capability-based systems; however, his work leaves off prior to later work emphasising micro-kernels and language run-time approaches [9].

Our hardware approach may be briefly summarised as differing from prior approaches in that it (a) adopts a RISC rather than microcode-centric philosophy, (b) occurs in the context of multi-threaded and multi-core processor design, (c) reflects contemporary concerns about software risk, (d) is able to build on (and hence respond to) a massive open source OS, programming language, and application corpus, and (e) adopts a hybridised design targeting incremental adoption in the context of existing software systems.

3.1 The capability system model

Capabilities are unforgeable tokens of authority that connote rights to underlying memory and objects. A *capability system* is one in which there is no *ambient authority*: rights of executing code are entirely captured by the set of capabilities explicitly delegated to it. The *object-capability security model* co-evolved with ideas about language type safety, object orientation, and particularly, encapsulation: access to private data associated with an object can be accessed only through invocation of its methods.

In a capability system, method invocation represents controlled security domain transition from caller to callee, and back—both parties may limit the sets of capabilities that flow in either direction, allowing both symmetric and asymmetric mistrust to be implemented. The mechanics of invocation vary dramatically by substrate, but fall into two general categories: an underlying message-passing substrate (micro-kernels and distributed systems), or a secure local method invocation mechanism grounded in type safety (hardware architectures and language run-times).

3.2 The CHERI architecture

In order to implement a hybrid capability system based on a revised hardware-software model, we have made a number of design choices, in many cases transposing similar choices from Capsicum to hardware:

1. Capability models are local to address spaces, placing a focus on implementing fine-grained compartmentalisation within applications and programming language run-times.
2. A capability register file supplements the general-purpose register file by representing a security context's working set of memory and object rights.
3. Instructions manipulating capabilities preserve safety properties such as memory protection. Wherever possible, capability inspection and manipulation is performed using safe but unprivileged hardware instructions without recourse to a supervisor.
4. CHERI supplements, rather than replaces, the 64-bit MIPS ISA and memory model, in order to support existing hypervisors, operating systems, language run-times, and applications. MIPS ISA instructions implicitly indirect through capability registers, as does instruction fetch. Hybrid applications are therefore supported: capability-aware libraries can be transparently invoked by capability-oblivious applications, and capability-aware applications can sandbox capability-oblivious code.

Figures 1 and 2 illustrate the high-level CPU and software architectures for CHERI. CHERI introduces an additional capability coprocessor[3] along-side the current CPU pipeline, which transforms MIPS ISA-originated operations, and implements new capability coprocessor instructions, prior to memory management unit (MMU) translation from virtual to physical addresses.

[3] In the parlance of MIPS, coprocessors are simply reserved portions of the ISA, and imply nothing about the implementation of packaging of additional logic. The MIPS ISA reserves coprocessor 0 for system control (configuration, MMU, and exception handling), and coprocessor 1 for floating point. We use the coprocessor 2 portion for the capability mechanism.

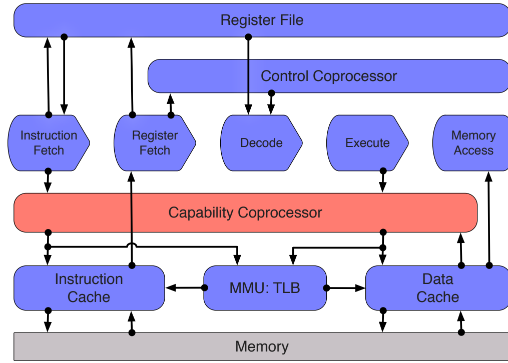

Figure 1. Placement of the CHERI capability coprocessor with respect to conventional CPU design elements

CHERI capability registers can describe both *memory segments* (bounded regions of virtual address space with controlled access properties) and *object capabilities*, which provide, with assistance from the compiler and software run-time, transitions between mutually untrusting protection domains. A hardware context's instantaneous rights are defined by the set of capabilities held in its register file; a software thread's protection domain is therefore described by the transitive closure of capabilities in its register file, capabilities it can fetch directly or indirectly via those capabilities, and the further capabilities it can retrieve through calls to the supervisor or invocations of object capabilities it can obtain.

We anticipate that a single UNIX process will contain many threads, each potentially executing with their own protection domain. A *thread context switch* is a *security domain context switch* within the capability model, and *object capability invocation* can be implemented as a supervisor exception handler in software, a dedicated instruction, or as *inter-thread message passing*. We argue that this correspondence between thread contexts and object invocation (previously observed in software capability systems such as Mach and Java) is particularly important in the hardware context, where recent work in processor design has provided low-latency hardware message passing techniques.

CHERI employs *tagged memory*, which allows the capability coprocessor to ensure the integrity of capabilities stored to, and fetched from, general-purpose memory[4]. CHERI capabilities are 256 bits long, plus a one-bit tag indicating whether a particular 256-bit line contains a capability, leading to a 0.4% space overhead. Capabilities may be intermingled with other memory, allowing them to be inserted (for example) into C data structures, or pushed onto and popped off of the stack.

3.2.1 Protection vs. virtualisation

CHERI's incremental adoptability goals require a clean composition not only between existing MIPS user applications and CHERI-aware code, but also between the existing MMU-based protection model and the capability model. With these goals in mind, and with explicit hopes to apply capability-based techniques in OS kernels, language run-times, and web browsers, we have selected a composition that places capability evaluation before virtual address translation. As a result, capabilities are interpreted with respect to specific virtual address spaces; i.e., within, rather than between, UNIX processes. When memory fetches and stores are issued via memory capabilities, they are bounds-checked, and then transformed into ambient virtual addresses for processing by the MMU. Con-

[4] It is possible to imagine a version of CHERI in which tagged memory is unsupported, and instead memory segments either hold capabilities or data.

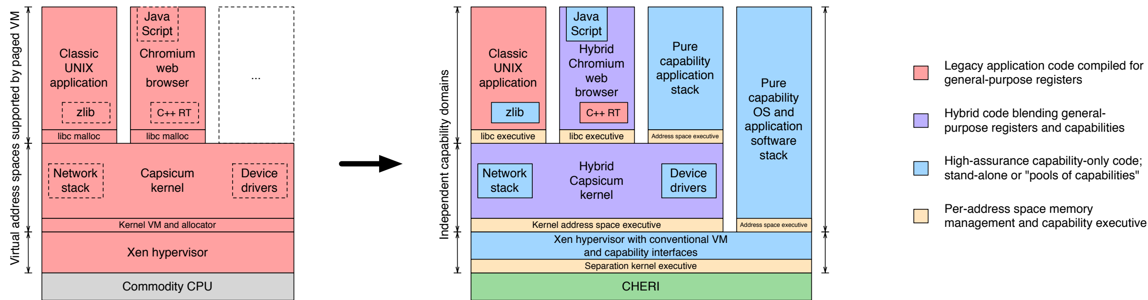

Figure 2. Before and after CHERI: a high-level system software architecture

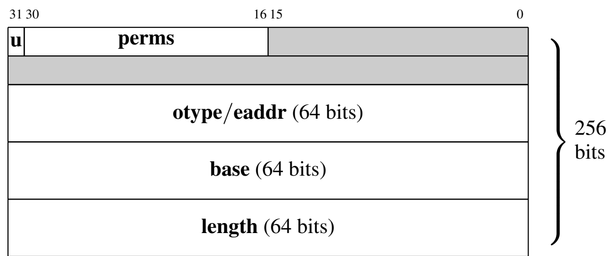

Figure 3. Contents of a capability

ventional MIPS ISA fetches and stores are automatically indirected via *capability zero*, and CPU instruction fetches are indirected via the *program counter capability*. This allows capabilities to be used to relocate and constrain unmodified MIPS code.

3.2.2 Capabilities

Each CHERI capability is a structured 256-bit entity, as shown in Figure 3. Each capability contains an unsealed bit (**u**), access permissions (**perms**), object type (**otype**/**eaddr**), base virtual address (**base**), and length in bytes (**length**). **perms** bits include general-purpose register fetch and store, object invocation and return, and capability register fetch and store. Capabilities may be held in capability coprocessor registers, or in tagged memory with 32-byte alignment. Programmers may reasonably think of capabilities as *fat pointers* referring to regions of memory or objects.

Memory capabilities have the **u** bit set, and describe a region of virtual address space and its protection properties using the **perms**, **base**, and **length** fields. General-purpose and capability registers may be fetched and stored via memory capabilities, subject to permissions and bounds. On a violation, an exception is triggered.

Object capabilities have the **u** (unsealed) bit cleared, and employ the **perms**, **otype**/**eaddr**, **base**, and **length** fields to describe a class and its instance data. Direct memory access to instance data is not permitted; instead, methods must be invoked using a special instruction. The **otype**/**eaddr** field allows the hardware to check, atomically, that an object and its class are linked; it may also be used for software type enforcement. However, hardware-level types should not be confused with language types, as they describe only the linkage between instance data and code to interpret it—there are no provisions for language-level concepts such as "inheritance", which are left to higher-level run-times.

Capability operations fall into several categories:

- manipulating capability registers, including copying fields to and from general-purpose registers,
- fetching and storing general-purpose and capability registers via a memory capability, and
- creating, invoking, and returning from object capabilities.

Instructions manipulating capabilities enforce a critical property: monotonically decreasing rights. Likewise, tagged memory prevents in-memory capabilities corrupted by accidental (or intentional) manipulation from being loaded into capability registers. Attempts to increase the rights on a capability, load a capability from non-capability memory, or improperly invoke a capability, lead to a hardware exception being thrown.

Object invocation is an operation for transitioning between two different protection domains, where the object capability reference held by the caller is a name for the callee. Callee instance data is unsealed during invocation, and may include additional capabilities to be used by the class. There are similar (but inverted) semantics for return. The compiler and linker must construct safe call and return code paths that not only save and restore rights, but also flush them appropriately in order to prevent undesired leaks of capabilities between caller and callee. We have spent a substantial amount of time exploring possible behaviours at object capability call and return, debating the split between hardware and software, whether or not to provide call-return semantics, and how the stack should behave.

3.3 Software models for capability use

CHERI supports a broad range of potential operational uses, from conventional hypervisor and OS designs to "pure" capability systems. However, its greatest strengths lie in the incremental deployment opportunities presented by its hybrid capability model. To this end, we have thus far investigated two software models:

1. Selective deployment of capabilities within a commodity code base: minimal changes to a commodity OS support application-layer use of capabilities, such as in the Chromium web browser.
2. Prototyping "pure" capability systems, which eschew MMU hardware, using capability features to sandbox capability-oblivious library and application code.

An important aspect of the CHERI model is that it is efficiently virtualizable, unlike ring-based schemes. A capability system in ring 0 is almost identical to running capability-based code on top of a hybrid operating system.

3.3.1 Memory model

With CHERI, memory capabilities name and connote rights to virtual address space—an intimate intertwining of memory model and protection model that implies, as with many managed language run-times, that the memory allocator is part of the TCB. Further, the CHERI model is optimised for enforcement without additional

memory indirection, and so contains no explicit support for revocation. We anticipate several software memory models being viable on CHERI, possibly even within a single address space:

- *Non-reuse of heap address space*. This model, proposed by Akritidis, relies on the effectively unbounded size of 64-bit address spaces[5], and has demonstrated security benefits [2]. Conventional virtual memory techniques can be used to enforce eventual revocation, subject to granularity and book-keeping.
- *Garbage collection of heap address space*. While CHERI has not specifically been designed for garbage collection, it should be supportable. Certain further ISA additions, such as forwarding pointers, might improve GC efficiency.
- *Limited pointer flow control enforcement*. An *ephemeral* bit in the **perms** mask allows for limited (two-colour) information flow control, added in anticipation that it would be used to temporarily delegate capabilities when invoking untrusted object capabilities. For example, this feature might be used to temporarily delegate caller stack references, ensuring that references could not subsequently be used from an asynchronous context.

Distinguishing allocation of memory for code, heap, and stack will be important in providing compatibility with existing code. Fortunately, widespread use of segment-centric memory models on x86, well-supported by current compilers and tool chain, will provide some assistance.

3.3.2 Future of the memory management unit (MMU)

As suggested by our hybrid capability system goals, we hope to support a diverse set of software models. At the low end of the spectrum, we wish to run entirely unmodified applications. Software will have a strong expectations of conventional virtual memory behaviour, and we anticipate that low-level hypervisors and operating systems, even if capability-aware, will continue to support conventional OS and process models. Here, the MMU remains central.

Moving along the spectrum, we are interested in hybridised libraries and applications that might be embedded within capability-unaware applications, and similarly, capability-based applications that sandbox capability-oblivious libraries. This model seems appropriate for application to our *most risky* software components, such as network stack packet processing and device drivers in the OS kernel, and data processing libraries (such as image decompression and video CODECs) in high-level applications such as web browsers and office suites. The MMU will continue to implement a process model, but with a decreased emphasis on protection. This should lead to greater virtual memory efficiency: use of the MMU to set up shared memory will be minimised, and page sizes will be able to grow due to avoidance of finer-grained protection.

At the high end of the spectrum, we anticipate code generation solely using CHERI instructions, with notions of ambient access to the address space entirely eliminated from most (or all) executing code. The TLB will retain three functions: enabling non-protection uses of virtual memory (e.g., swapping), revocation of portions of the virtual address space, and virtualised capability machines. TLB aliasing should rarely or never occur, and most page sizes should be extremely large, improving memory access performance.

3.3.3 C language model

Source code and binary compatibility with existing C-language TCBs are key design goals for CHERI. We have formulated initial C bindings for capabilities, with the awareness that C currently omits language-level support for threads—a striking omission even without our thread-centric security model. We plan to define a capability-aware ABI and calling conventions, and allow data types to be annotated at a language level for code generation using memory capabilities. Arbitrary pointer arithmetic cannot be expressed on capabilities themselves, but can be expressed within a capability-named region of memory. We hope to use annotations along the lines of `__capability__` to request use of memory capabilities, and `__ephemeral__` to specify ephemeral arguments.

We must then determine how to capture the notion of object capabilities within C. Other capability-esque, managed programming language run-times, such as Java and for CLR, have opted to align language-level object features with bytecode protection models. As C does not have language-level support for objects, we instead plan to make use of an explicit API for object invocation and return, although with compiler support in order to ensure appropriate interactions between language and hardware features.

4. Implementation

We are now beginning the second year of the CTSRD project, which includes BERI and CHERI. In this section, we discuss, our implementation work, and results, to date.

4.1 Bluespec Experimental RISC Implementation (BERI)

In order to scientifically answer the question, "does protection belong in hardware or software", we first need a hardware research platform capable of expressing a variety of protection models. We also require an experimental software stack that captures the complexity of real-world compartmentalised applications spanning operating systems, programming languages, and applications. Open source software systems such as FreeBSD, Linux, X.org, Apache, Python, Chromium, OpenOffice, etc, offer realistic proving grounds for software experimentation. However, there is a clear gap with respect to experimental hardware platforms.

To this end, we have developed a Bluespec Experimental RISC Implementation (BERI)—a parameterisable 64-bit MIPS FPGA soft core. BERI has many of the attributes of a contemporary CPU design, including:

- A pipelined processor design with multiple levels of cache
- 64-bit MIPS ISA-compatible registers and full range of ALU, branch, and control instructions
- System control coprocessor including an MMU, exception handling, and multiple CPU protection rings

We have omitted certain features that do not pertain directly to this research, including 32-bit MIPS compatibility mode, little-endian mode, and a floating-point coprocessor. Whereas we have an immediate interest in multi-threaded and multi-core designs, we have not implemented superscalar features. This both reduces the complexity of our implementation, and reflects a trend in hardware design towards exposing parallelism explicitly through multi-threading and vector instructions, rather than speculative execution.

Unlike existing research soft cores, BERI is implemented in Bluespec, a high-level hardware description language (HDL) based on Haskell [4]. In contrast traditional HDLs (such as Verilog and VHDL), Bluespec allows for typed modular abstraction and highly parameterisable designs, as well as easier design space exploration; among other features, Bluespec design specifications describe atomicity rather than timing properties of logic. As a result, we are able to easily change key parameters of the processor, and able add new features with (relative) ease. For example, we can experiment with different TLB sizes, introduce features such as multi-threading and hardware page table walking, add a capability coprocessor, or even omit the MMU entirely—all key experimental variables to explore the balance between hardware and software.

[5] "[64 bits] ought to be enough for anyone."

Sandbox 0: drawing application

~140 lines of conventional C code compiled to 64-bit MIPS

Sandbox 1: footer bar

~90 lines of conventional C code compiled to 64-bit MIPS

Sandboxed user library code

~600 lines of conventional C code compiled to 64-bit MIPS: memcpy, memset, strlen, printf, framebuffer, touch screen

~40 lines of inline MIPS and CHERI assembly: framebuffer, touch screen

Deimos microkernel

~1800 lines of conventional C code compiled to 64-bit MIPS: trusted path, device drivers, diagnostics

~700 lines of CHERI-specific C code: capability management, context switching

~450 lines of MIPS and CHERI ISA assembly: bootstrap, exception handling, capability management

CHERI prototype

~10,500 lines of Bluespec

Figure 4. Deimos: demonstration operating system from Mars

We are able to run a significant quantity of 64-bit MIPS code, and are porting the FreeBSD operating system and Chromium web browser to provide a more complete experimentation platform. Strong support for open source software is key; we will merge any changes required in order to provide a long-term research platform upstream to FreeBSD. We plan to open source BERI in 2012; this will not only allow our experiments to be reproduced, but also facilitate hardware-software interface research elsewhere.

4.2 The CHERI prototype

We have been quite successful in the development of a Bluespec-based research platform. We began with an experimental multi-threaded MIPS system (MAMBA) developed by Gregory Chadwick, and have significantly fleshed out its functional completeness and correctness. We have pipelined the CPU, added a system control coprocessor, and developed an extensive test suite. The current prototype can be run in pure software simulation, or synthesised to two different Altera-based FPGA teaching boards, the Terasic DE-4 and tPad platforms.

We have implemented a prototype capability coprocessor, and extended the GNU assembler, `gas`, to support these features. We have not yet implemented tagged memory, nor hardware-accelerated object capability invocation. MAMBA's multi-threading support has been temporarily removed during pipelining work, but we hope to restore this in the near future, at which point we will be able to investigate a variety of techniques for accelerating non-hierarchical protection domain transition.

4.3 Deimos: demonstration operating system from Mars

In order to explore our ideas about hybrid capability models early in the project, we have constructed a prototype capability-based micro-kernel operating system, Deimos. Deimos uses the capability coprocessor (CP2) to implement sandboxing and delegation, rather than the MMU, exercising memory capabilities, exception handling, and a broad range of general-purpose MIPS features—all within the CPU's kernel protection ring.

We have constructed a demonstration scenario based on the Terasic tPad. The demo executes in three protection domains: a fully privileged micro-kernel, and two sandboxes, one implementing a touch screen-based drawing application, and a second presenting independent image content to the user on another portion of the screen. Screen access is delegated to applications using memory capabilities for portions of a hardware frame buffer, illustrating the flexibility of the capability model. As shown in Figure 4, roughly 85% of demonstration code is general-purpose C code compiled using an unmodified MIPS `gcc`. The remainder consists of capability-aware C source code using inline assembly.

This experiment has proven interesting in a number of ways—Deimos makes use of memory capabilities for the purposes of relocating capability-unaware MIPS code within a capability-aware operating system, exploring one of the more extreme ends of the hybrid OS spectrum. It also illustrates that delegation of shared memory resources, and even hardware resources, is straightforward using a capability mechanism. Without hardware support for tagged memory in our current Bluespec prototype (roughly 10,500 lines), we do not allow storing and loading capabilities in sandboxes; however, the OS saves and restores capability and general-purpose register state when switching, and OS system calls carry capability as well as general-purpose values using the CHERI ABI. In many ways, the Deimos software environment resembles what an in-process environment might look like on a UNIX system running in a hybrid capability mode: a virtual address space containing both capability code with ambient authority, and unmodified MIPS code running in capability sandboxes, able to interact with other components across security boundaries with minimal hybrid shims.

4.4 CheriBSD

We hope to use FreeBSD as a platform for larger-scale experiments in fine-grained compartmentalisation, both within the OS kernel (e.g., for device driver sandboxing), and within applications (e.g., within the Chromium web browser). As with conventional floating-point coprocessors, we anticipate that a small set of changes to detect use of the coprocessor, conditionally save/restore its state, and ensure a consistent execution environment for the kernel, will allow immediate userspace experimentation with capabilities while leaving the kernel otherwise unmodified.

We will then deploy capability support in low levels of the FreeBSD runtime linker and `libc`. We will define ELF and linker extensions necessary to set up appropriate linkage in and between capability-aware and capability-oblivious code, allowing experimentation with new protection features inside of critical code bases.

We anticipate a variety of interesting problems—not least how to handle the relationship between a conventional OS kernel and modified protection environment within processes it hosts. The question, "What code is allowed to perform a system call?" hints at the subtlety in even initial experiments. Another interesting question is how to safely and usefully handle exceptions in the presence of multiple protection domains.

5. Proposed evaluation techniques

With prototyping efforts still underway, evaluation of the CHERI approach remains premature. However, our early work in developing Deimos has proven begun to validate our hypothesis that a hybrid capability model may meet our short-term and long-term compatibility goals. In this section, we consider potential evaluation approaches that we hope to explore in the coming year.

5.1 Viability of hybrid software models

Success of the CHERI approach will depend on providing software models that are easy to program to, optimise for, and debug. One key aspect of this is our hybrid design argument: if all current C code can be reused largely unmodified, then CHERI will prove at least as accessible as current systems. However, we presuppose that some change is necessary in order to allow programmers to declare software compartmentalisation properties—this is a known hard problem, and one of the motivations for CHERI.

Easy-to-use C language bindings will be critical; we do not anticipate achieving the simplicity of Java, in which substrate protection properties are derived directly from language-level properties.

However, we hope that allowing generation of memory capability-enabled code from C annotations, and providing explicit access to protection domain transitions at the language level, will be a dramatic improvement over current MMU-based models of compartmentalisation. Providing a single address space for compartmentalised applications should ease debugging and program analysis.

A range of interesting questions might be asked given these foundations. What sorts of changes are required to critical libraries and applications to see security improvements? Are changes to applications sufficient simple and compatible that they can be maintained and upstreamed to application providers without encountering significant portability problems?

5.2 Performance comparison of protection approaches

A central research question being addressed by our work is whether protection features for low-level TCBs are most efficiently implemented via current CPU protection models, pure software enforcement models, new capability-based models, or hybridisations of these approaches. The CHERI hardware architecture is designed specifically to allow testing such hypotheses through not only its hybrid capability model, but also our ability to include and exclude aspects of the CPU (such as the MMU, capability coprocessor) from the design, as well as vary parameters (such as TLB size) in our experiments. To this end, supporting a single source code representation of protection and allowing back-end targets to be substituted for one another will be important. One possible direction will be to implement a CHERI-like model using techniques similar to Google NaCl and PNaCl in order to directly compare software vs. hardware-assisted enforcement considerations.

5.3 Energy use of different protection models

RISC-derived systems see their greatest deployment in low-energy environments, with power efficiency being a key argument for ARM, and to a lesser extent MIPS, CPUs. A fascinating question is how the distribution of protection and enforcement across compile-time checking, run-time static analysis, dynamic enforcement using software, and dynamic enforcement using hardware protection features, affects energy use. One intriguing argument is that dynamic checking associated with protection features is a closely intertwined but parallel computation to the actual computation desired by the programmer. If hardware parallelism can be exploited to compute protection concurrently with the base computation, then the clock rate could be reduced, improving power efficiency.

6. Future work

This workshop paper has captured CHERI at an intermediate stage—one in which we have an increasingly viable research platform for investigating issues in the hardware-software interface, and early prototyping results for hybrid capability operating systems. Our immediate plans are to complete the prototype:

1. Fill remaining functional gaps in BERI and FreeBSD to allow a mainstream mobile-, workstation-, and server-class operating system to reach multi-user mode; port, as required, useful application software such as Apache and Chromium.
2. Adapt FreeBSD to operate as a hybrid operating system able to host capability-based libraries and applications. These changes should be quite small: extensions to exception handling and context switching in FreeBSD so that capability state is saved and restored, and to allow the FreeBSD kernel to limit access to system calls to authorised userspace components.
3. Complete development of a capability-enhanced ABI, exception model, and tool chain for 64-bit MIPS. Our informal ABI must be formalised, and core system components (such as the runtime linker and C library) must use this ABI effectively. We hope to locally adapt first the 64-bit MIPS LLVM back-end, and later the LLVM IR, to support CHERI features.
4. Begin a performance measurement and optimisation cycle, in particular to explore efficient implementations of object capability invocation; for example, investigate whether observed congruence between invocation and inter-thread message passing offers potential performance wins. Treating parameters such as capability register namespace size, TLB size, cache size, and other historically static elements of CPU design as independent variables, enabled by the BERI processor design, will allow us to explore a multi-dimensional space. We may also implement an optional hardware page-table walker for BERI—a traditional design element in x86 and ARM, but not MIPS, CPUs—in order to more cleanly explore the design space.

The BERI research platform will allow us to grapple, for the first time, with hardware-software interface research issues that have not been easily accessible before. As such, we hope to apply it to a broad range of research problems beyond security.

7. Related work

The question of placement of protection in hardware or software has been considered at many times in computing history, and certainly bears further investigation at this moment in which security risks have become so critical. Several recent efforts have explored alternatives, often grounded in protection look-aside buffers (PLBs), which provide finer-grained protection controls than TLBs. Designs such as the Mondrian model and Legba fall into this category, and have largely been evaluated using simulation [23, 25]. CHERI takes a markedly different approach by making protection an explicit part of the compiler's model through capabilities.

Other recent research into alternative protection models includes the TIARA design, and a derived approach in the in-progress SAFE architecture [5, 20]. These systems integrate notions of information flow and extremely granular tagging at a word level with software-defined policy, in contrast to CHERI's more fixed capability model using only a single bit per line for tagging. We hope to perform more detailed comparisons of the CHERI and SAFE architectures as they mature; while both can clearly express a full range of protection models, key aspects for consideration are the performance, ease of expression, and adoptability.

8. Conclusion

This paper summarizes the current design and development of CHERI, the first *hybrid capability model* CPU architecture, able to run both capability-based and commodity application and library code side-by-side. In order to properly evaluate this approach, we have designed and implemented BERI, a research platform for the hardware-software interface. Despite the overall project being a work-in-progress, initial results such as the development of a CPU capability-based micro-kernel, Deimos, appear promising.

Using this platform, we hope to investigate more critical questions in computer architecture, requiring a whole-system view: should fine-grained protection be differentiated from virtualization in order to improve scalability, programmability, and performance? Can a hybrid capability model (which combines hardware capability features with a commodity memory management unit design offer) both performance and compatibility in order to provide an incremental adoption path for fine-grained protection within current C-language TCBs? More fundamentally, does fine-grained protection require hardware assistance for reasons of performance and energy use? This paper provides some initial hints in this direction,

and seeks the feedback of a large community to ensure that our coming research results are of greatest use.

Acknowledgments

We would like to thank our colleagues—especially Gregory Chadwick, Rance DeLong, Steven Hand, Patrick Lincoln, Anil Madhavapeddy, Andrew Moore, Robert Norton, and John Rushby, our summer students, Wojciech Koszek, Ilias Marinos, and Will Morland, and members of our external oversight group.

The CTSRD Project gratefully acknowledges the support of its research sponsors. This work was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense. Portions of this work were supported by Google, Inc.

References

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

[2] P. Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[3] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 309–322. USENIX Association, 2008.

[4] *Bluespec SystemVerilog Version 3.8 Reference Guide*. Bluespec, Inc., Waltham, MA, November 2004.

[5] A. DeHon, B. Karel, J. Thomas F. Knight, G. Malecha, B. Montagu, R. Morisset, G. Morrisett, B. C. Pierce, R. Pollack, S. Ray, O. Shivers, J. M. Smith, and G. Sullivan. Preliminary design of the SAFE platform. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems (PLOS 2011)*, October 2011.

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

[7] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[8] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an operating-system kernel. *Commun. ACM*, 53:107–115, June 2009.

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

[10] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek. Software fault isolation with API integrity and multi-principal modules. In *SOSP 2011: Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011.

[11] S. McCanne and V. Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In *USENIX'93: Proceedings of the USENIX Winter 1993 Conference*, Berkeley, CA, USA, 1993. USENIX Association.

[12] A. Mettler, D. Wagner, and T. Close. Joe-E: A Security-Oriented Subset of Java. In *NDSS 2010: Proceedings of the Network and Distributed System Security Symposium*, 2010.

[13] P. Neumann and R. Feiertag. PSOS revisited. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003), Classic Papers section*, pages 208–216, Las Vegas, Nevada, December 2003. IEEE Computer Society.

[14] P. G. Neumann and R. N. M. Watson. Capabilities revisited: A holistic approach to bottom-to-top assurance of trustworthy systems. In *Fourth Layered Assurance Workshop*, Austin, Texas, December 2010. U.S. Air Force Cryptographic Modernization Office and AFRL.

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

[16] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *EuroSys '09: Proceedings of the 4th ACM European Conference on Computer Systems*, pages 219–232, New York, NY, USA, 2009. ACM.

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

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

[19] J. Shapiro, J. Smith, and D. Farber. EROS: a fast capability system. In *SOSP '99: Proceedings of the seventeenth ACM Symposium on Operating Systems Principles*, Dec 1999.

[20] H. Shrobe, T. Knight, and A. de Hon. TIARA: trust management, intrusion tolerance, accountability, and reconstitution architecture. Technical Report MIT-CSAIL-TR-2007-028, Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Lab, May 2007.

[21] R. Wahbe, S. Lucco, T. E. Anderson, and S. u. L. Graham. Efficient software-based fault isolation. In *SOSP '93: Proceedings of the fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, New York, NY, USA, 1993. ACM.

[22] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical capabilities for Unix. In *Proceedings of the 19th USENIX Security Symposium*. USENIX, August 2010.

[23] A. Wiggins, S. Winwood, H. Tuch, and G. Heiser. Legba: Fast hardware support for fine-grained protection. In A. Omondi and S. Sedukhin, editors, *Proceedings of Advances in Computer Systems Architecture, 8th Asia-Pacific Conference, ACSAC 2003*, volume 2823 of *Lecture Notes in Computer Science*, pages 320–336. Springer, 2003.

[24] M. Wilkes and R. Needham. *The Cambridge CAP Computer and Its Operating System*. Elsevier North Holland, New York, 1979.

[25] E. Witchel, J. Rhee, and K. Asanovic. Mondrix: Memory isolation for linux using mondriaan memory protection. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

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

[27] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, Washington, DC, USA, 2009. IEEE Computer Society.