The CHERI capability model: Revisiting RISC in an age of risk

Jonathan Woodruff[†] Robert N. M. Watson[†] David Chisnall[†] Simon W. Moore[†] Jonathan Anderson[†]
Brooks Davis[‡] Ben Laurie[§] Peter G. Neumann[‡] Robert Norton[†] Michael Roe[†]
[†] University of Cambridge [‡] SRI International [§] Google UK Ltd
firstname.lastname@cl.cam.ac.uk {neumann,brooks}@csl.sri.com benl@google.com

Abstract

Motivated by contemporary security challenges, we reevaluate and refine capability-based addressing for the RISC era. We present CHERI, a hybrid capability model that extends the 64-bit MIPS ISA with byte-granularity memory protection. We demonstrate that CHERI enables language memory model enforcement and fault isolation in hardware rather than software, and that the CHERI mechanisms are easily adopted by existing programs for efficient in-program memory safety.

In contrast to past capability models, CHERI complements, rather than replaces, the ubiquitous page-based protection mechanism, providing a migration path towards deconflating data-structure protection and OS memory management. Furthermore, CHERI adheres to a strict RISC philosophy: it maintains a load-store architecture and requires only single-cycle instructions, and supplies protection primitives to the compiler, language runtime, and operating system.

We demonstrate a mature FPGA implementation that runs the FreeBSD operating system with a full range of software and an open-source application suite compiled with an extended LLVM to use CHERI memory protection. A limit study compares published memory safety mechanisms in terms of instruction count and memory overheads. The study illustrates that CHERI is performance-competitive even while providing assurance and greater flexibility with simpler hardware.

1. Introduction

Research systems such as Mondrian Memory Protection [45, 46] and Hardbound [12], and industrial approaches such as Intel's recently announced Memory Protection Extensions (iMPX) [17], have shown the importance of architectural support for fine-grained memory protection. Mondrian in particular identified the conflation of protection with translation as a flaw in existing approaches: paging is useful for operating systems that provide coarse-grained separation and virtualization (*translation*), whereas segmentation is more useful to enforce intra-program *protection*. Program safety and security depends on enforcing *pointer safety* (the size and permission aspects of dynamic *type safety*) and *isolation* (for *sandboxing* or *application compartmentalization* [40]).

Capability-system proponents have long argued that a strong underlying protection model can improve software robustness and security [11, 42]. However, limited historical demand for fine-grained protection, combined with significant technical challenges (especially compatibility), has challenged the adoption of capability systems. In contrast, coarse-grained virtual-memory protection has seen wide deployment to isolate application instances from one another. Ubiquitous networking and widespread security threats have renewed interest in finer-grained protection models that not only improve software debuggability, but also mitigate vulnerability exploit techniques (e.g., code injection via buffer overflows).

Processors with capability-based addressing, epitomized by designs such as the M-Machine [5], provide strong technical and intellectual grounding for intra-program protection. However, such systems fail to provide adequate compatibility with existing source-code and binary software corpora, and often require ground-up software rewrites. In contrast, hardware and software bounds-checking techniques often exchange safety (and sometimes performance) for clearer adoption paths.

In this paper, we introduce Capability Hardware Enhanced RISC Instructions (CHERI), a *hybrid capability model* that blends conventional ISA and MMU design choices with a capability-system model. Key features include a *capability coprocessor* (defining a set of compiler-managed *capability registers* holding capabilities similar to unforgeable segment descriptors), and *tagged memory* (protecting in-memory capabilities). Capability addressing occurs before virtual-address translation such that each process is a self-contained virtual capability system. As programs adopt the CHERI ISA, object accesses use these registers to check bounds, control access, and protect pointer integrity. Tagging allows data and capabilities to be safely combined within data structures. CHERI offers scalable and secure intra-address-space protection, and retains a high level of source-code and binary compatibility. While the 64-bit MIPS ISA was our starting point, the approach is not specific to MIPS. Our key contributions are:

- a novel hybridization of capability-based addressing with a RISC ISA and MMU-based virtual memory emphasizing both performance and compatibility;
- an FPGA-based microprocessor implementation demonstrating our approach;
- adaptations of the LLVM compiler suite [22] and FreeBSD operating system [26] to use these features;
- functional and simulation-based comparisons of conventional and research protection models; and
- benchmark comparison with software bounds checking.

978-1-4799-4394-4/14/$31.00

We first describe requirements for fine-grained memory protection, our RISC memory capability model, its realizations in the 64-bit MIPS ISA, and implementation on FPGA. We then compare the CHERI model with a number of other research and fielded protection models, exploring tradeoffs in protection, compatibility, and performance. A limit study, derived from execution traces, reveals that CHERI has competitive performance with other models. We also run benchmarks on our FPGA implementation in order to understand practical implementation and performance considerations, which illustrate significant improvements over software bounds-checking.

2. Practical memory protection requirements

Practical userspace protection has several desirable properties:

Unprivileged use Protection should be the common case and therefore should not require frequent system calls.

Fine granularity Granularity should accommodate data structures that are small and densely packed (e.g., on-stack), or with odd numbers of bytes or words.

Unforgeability Software should not be able to increase its permissions, accidentally or maliciously.

Access control Hardware should enforce region permissions such as store and execute.

Segment scalability Performance and memory storage overhead should scale gracefully with the number of protected memory regions.

Domain scalability Performance and memory storage overhead should scale gracefully with the number of protection domains and frequency of their communication.

Incremental deployment Extant userspace software should run without recompilation even as selected components, such as shared libraries, make use of fine-grained protection.

Furthermore, we believe userspace protection should exploit program knowledge to offer pointer safety, not just address validity. *Address validity* models associate protection properties with regions of address space. Paged virtual memory is an address validity mechanism. *Pointer safety* models associate protection properties with object references. Fat pointers are a pointer safety mechanism. Pointer safety is more precise than address validity and can, for example, distinguish between a buffer overflow and a reference to an adjacent object in memory. Address validity, however, makes supervision more convenient due to a centralized protection table, and enables features such as efficient revocation.

We observe that pointer safety implies a segmented view of memory rather than the common flat view. It should be possible to blend these two views on memory to gain safety without unnecessarily breaking compatibility.

3. A RISC memory capability model

While the requirements outlined above are complex, the hardware mechanism that fulfills these requirements should be as simple as possible to ease adoption by processor manufacturers. The most efficient unforgeable pointer safety implementations use a *memory capability model*. In such a model, a *memory capability* is an unforgeable pointer that grants access to a linear range of address space. To maintain safety, all memory accesses must occur through a memory capabilities.

It is possible to implement a memory capability model in a strict RISC instruction set with a load-store architecture and single-cycle operations as demonstrated by the M-Machine [5]. In a RISC implementation, memory capabilities can be stored in registers or in memory, but must be loaded into registers for use. The current *protection domain* is defined by the capabilities stored in registers along with all capabilities in memory reachable through those capabilities. With explicit management of memory capabilities in the style of pointers, there is no need for an associative table (such as the protection lookaside buffer in Mondrian) for managing permissions. This approach allows protection to scale with memory space rather than with a fixed resource like the TLB [27]. This approach also allows performance with protection to match a general-purpose pointer model.

To meet our memory-protection requirement in a RISC memory capability architecture, we must ensure that:

1. Capability manipulation instructions are unprivileged.
2. Capabilities can span any range in the virtual address space.
3. Legacy references are supported, but are constrained by the capability memory model.

These constraints make a RISC capability model not only efficient, but useful to modern software stacks.

4. CHERI implementation

We have chosen to implement a RISC capability model as an extension to the 64-bit MIPS IV instruction set. MIPS has a well-established 64-bit specification and adheres to a prototypical RISC philosophy. We implemented the processor in Bluespec SystemVerilog [4], with a general parity of features with the MIPS R4000. As with the MIPS R4000, our base processor (Bluespec Extensible RISC Implementation (BERI)) is single-issue and in-order, with a throughput approaching one instruction per cycle. BERI has a branch predictor and uses limited register renaming for robust forwarding in its 6-stage pipeline. BERI runs at 100MHz on an Altera Stratix IV FPGA and is capable of running the stock FreeBSD 10 operating system and associated applications. The CHERI processor adds capability extensions to BERI and is fully backward compatible, facilitating side-by-side comparisons.

CHERI capability extensions are implemented as a MIPS coprocessor, CP2. Similar to the MIPS floating-point coprocessor, CP1, the capability coprocessor holds a new register file and logic to access and update it. The MIPS pipeline feeds instructions into the capability coprocessor, exchange operands with it, and receive exceptions from it. The capability coprocessor also transforms and limits memory requests from instruction fetch and MIPS load and store instructions.

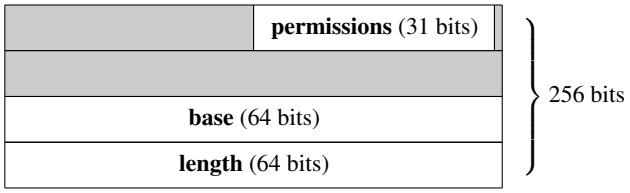

Figure 1: Memory capability

4.1. Capability registers

CHERI implements an additional register file for capabilities. This approach distinguishes capability state from integer state (and floating point state) in the architecture to avoid dynamic register types. There are 32 capability registers, each 256-bit wide, mirroring the number of integer and floating-point registers in MIPS. A commercial implementation might consider a smaller register set that would not unduly increase stack spills, and would reduce context-switch overhead and hardware resources, but we have maintained a large set for experimentation and for consistency with the MIPS architecture.

The currently implemented capability structure is shown in Figure 1. The **base** and **length** fields are the two basic fields needed to describe a segment of memory. We have allocated 64 bits to each, and choose not to implement a compression algorithm at this time to allow maximum flexibility as a research tool. The **permissions** field is a 31-bit vector with a "1" in each position indicating an allowed permission for the region. Permissions include load data, store data, execute, and load and store for capabilities. The other 26 permissions, and remaining capability fields, are being used for experimentation as described in Section 11. An implementation intended for widespread deployment would likely use a denser representation – for example, 128-bits using 40-bit virtual addresses or the Low-Fat Pointer approach [20].

Existing MIPS load and store instructions are implicitly offset via capability register 0, **C0**, and instruction fetches are offset via an implied program counter capability, **PCC**. This allows legacy code to run unmodified on CHERI, facilitating incremental adoption, and also allows sandboxing of unmodified programs within a parent address space.

We have also added a full set of load and store operations for addressing memory through capability registers with both immediate and register offsets. As MIPS lacks native register-indexed addressing, capability-relative addressing can often be faster than legacy loads and stores.

4.2. Capability manipulation and protection

The greatest challenge for a protection model is to protect memory capabilities from arbitrary manipulation (unforgeability) without appealing to the kernel (unprivileged use). This is important, as system calls remain a relatively expensive operation. For example, `malloc()` implementations typically amortize kernel entry by using a single `mmap()` system call to acquire a large block of memory for disbursement over many allocations [13]. A memory protection scheme that requires a system call for every `malloc()` would negate this optimization and be avoided in performance-sensitive applications.

Mnemonic	Description
CGetBase	Move base to a GPR
CGetLen	Move length to a GPR
CGetTag	Move tag bit to a GPR
CGetPerm	Move permissions to a GPR
CGetPCC	Move the PCC and PC to GPRs
CIncBase	Increase base and decrease length
CSetLen	Set (reduce) length
CClearTag	Invalidate a capability register
CAndPerm	Restrict permissions
CToPtr	Generate **C0**-based integer pointer from a capability
CFromPtr	CIncBase with support for NULL casts
CBTU	Branch if capability tag is unset
CBTS	Branch if capability tag is set
CLC	Load capability register
CSC	Store capability register
CL[BHWD][U]	Load byte, half-word, word or double via capability register, (zero-extend)
CS[BHWD]	Store byte, half-word, word or double via capability register
CLLD	Load linked via capability register
CSCD	Store conditional via capability register
CJR	Jump capability register
CJALR	Jump and link capability register

Table 1: CHERI instruction-set extensions

To preserve capability integrity while allowing user-space management, we must restrict capability manipulation, particularly in memory. That is, capabilities in memory must not be corrupted by general-purpose stores. Some traditional capability machines [42] and capability microkernels such as seL4 [19] have done this by defining regions of memory that can store capabilities distinct from those that can store data. This approach is problematic for a fat-pointer approach, as most contemporary programming languages allow arbitrary intermixing of pointers and data. In keeping with the RISC idea of the ISA as a compiler target [29], we have implemented *tagged memory* rather than supporting only regional separation. Valid capabilities are identified by an extra 'tag' bit associated with each 256-bit location. Any non-capability store clears this bit, thereby protecting the integrity of capabilities in memory without appealing to kernel mode.

With capabilities protected in memory, we implement user-mode instructions to safely manipulate capabilities in the register file. Table 1 shows a summary of the instructions that CHERI adds to the MIPS IV ISA. These include a full complement of load and store instructions, instructions for inspecting

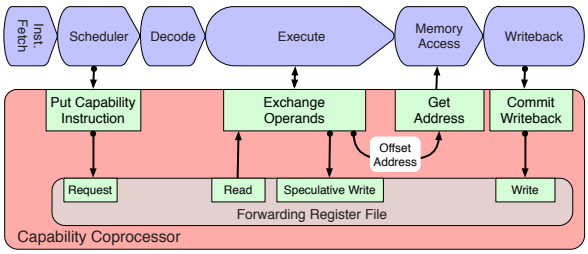

Figure 2: BERI pipeline with capability coprocessor

a capability, and for reducing (but not extending) the rights granted by a capability. Instructions that change fields in a capability must strictly reduce privilege, that is, disclaim permissions or reduce the extent. These restrictions allow CHERI to ensure capabilities are *unforgeable*. With the software unable to fabricate arbitrary memory references, a *protection domain* is defined by the transitive closure of memory capabilities reachable from its capability register set. Under an operating system, a process that begins with a capability for all privilege to its virtual address space can construct arbitrarily restricted domains described by unforgeable references.

CHERI tags physical memory, not virtual memory, and therefore maintains a single table for the entire system. This table holds one tag bit for each 256-bit line in memory, or 4MB of tag space per gigabyte of memory. A tag manager below the last level cache presents a 257-bit, tagged-memory interface to the CHERI cache hierarchy. The manager associates each memory transaction with a tag from the table and ensures consistency between memory and tags. The CHERI cache hierarchy propagates capability tags and implements CHERI tag semantics (which preserve the tag for a capability store and clear a tag on a general-purpose store). The decision to use physical – not virtual – memory for tags eliminates translation for the tag table (as required by Hardbound), and allows the tags to accompany physical cache lines through the cache hierarchy. CHERI allows capability registers to contain general-purpose data, which preserves the cleared tag to prevent use as a capability. This allows capability load and store instructions to copy 256-bit blocks of memory while remaining oblivious to whether they are copying data or a capability. As a result, a simple implementation of `memcpy()` can copy data structures containing both.

Our prototype maintains the tag table in DRAM. We could alternatively move it to a smaller memory that can be accessed in parallel with DRAM, or store tags in ECC-like bits to eliminate table lookups. However, the current tag controller (which minimizes table lookups using an 8KB tag cache) does not noticeably degrade performance.

4.3. Compatibility

CHERI allows capability-aware and legacy code to share an address space. Unsandboxed legacy executables run with access to the full address space, but may invoke capability-protected libraries. For example, an unmodified web browser can invoke capability-aware image libraries or video CODECs via MIPS-ABI functions. Capability-aware code can use sandboxed legacy code by restricting the default instruction and data capabilities (**PCC** and **C0**). The `CToPtr` and `CFromPtr` instructions convert between C pointers and capabilities to support safe and efficient interaction between capability-aware and legacy code.

The CHERI capability model requires minimal support from the OS. CHERI capabilities are layered atop standard paging, so the virtual memory system works without modification. On CPU reset, capability registers are initialized, granting the OS access to the entire address space so an OS can run unchanged without knowledge of the capability extensions. Indeed, we first achieved stability with unmodified FreeBSD on the processor before we added support for capabilities.

Our extended version of FreeBSD enables the capability coprocessor on boot; when the first user process is created or *execve()* is invoked, the entire user virtual address space is delegated to the user register file. The kernel saves and restores per-thread capability-register state on context switches. The user process then manages capabilities within that space, thus restricting access. Capability-aware allocators can manage memory and return capabilities in much the same way as conventional memory allocators. Revocation can be accomplished via zero-address-space-reuse allocators, TLB unmapping, or by a simplified version of garbage collection (made reliable by capability tags). New TLB permissions authorize capability loads and stores. The OS virtual-memory system is being extended to preserve tags for swapped pages.

4.4. Pipeline organization

As seen in Figure 2, our capability extensions are modularized as a MIPS coprocessor with a dedicated register file. All data accesses reference a capability register either explicitly or implicitly (**C0**), so the capability coprocessor is tightly coupled with the Execute and Memory Access stages of the pipeline. While instruction fetches are logically offset by a capability register, **PCC**, in implementation CHERI uses an absolute address for the program counter and validates it against **PCC** in the Execute stage to simplify both forwarding and instruction address calculation.

The capability register file is an instantiation of the general-purpose forwarding register file and inherits register renaming. All capability manipulation instructions are single cycle, as are capability loads and stores. This style of manipulation has orders of magnitude higher performance than protected segment manipulation on IA32 that, for example, required at least 241 cycles on a 1.1GHz Pentium III [21].

5. Use cases for CHERI capabilities

CHERI user-managed memory protection has a variety of potential uses, most of which are unexplored in contemporary operating systems. Cheap, fine-grained memory protection radically changes the memory-safety trade-off topography, and

we hope that our platform will enable exploration of this new landscape. This section describes some anticipated use cases.

5.1. Memory safety for C

The C language provides programmers with a great deal of flexibility, but the price of this flexibility is a lack of memory safety. Previous work has attempted to provide C programmers with greater memory safety via static and dynamic checks in software. Cyclone [18] is a variant of C that explicitly allows definition of fat pointers that are dynamically checked; CCured [28] automates the same process with either static verification or run-time checks. Fat pointers come at a run-time cost, but adding hardware support for fat pointers in the form of CHERI capabilities removes most of the distribution and enforcement costs.

CHERI capabilities can be used as general-purpose pointers with the limitation that their range cannot be enlarged. We have extended the LLVM [22] compiler framework and Clang, the C front end, to implement pointers as memory capabilities to ensure they are used according to programmer intent, including bounds checking and read, write, and execute property enforcement. We have added support for the `__capability` and `__output` qualifiers in the Clang front end and extended its understanding of casts. We improved LLVM's notion of address space to encode capabilities, and added special cases for a small number of optimizations that assumed that pointers were integers. Finally, we extended the MIPS back end with support for our new instructions. As a result, a `malloc()` that returns a capability will use the **`CIncBase`** and **`CSetLen`** instructions to construct a capability for the region that can be used to address the object with automatic bounds checking. In addition, a **`const`**-qualified capability pointer will explicitly disclaim the write permission via the **`CAndPerm`** instruction, so that the processor will throw an exception if attempts are made to write through it. We anticipate that other languages will make similar use of this functionality, but aim to provide a rich toolbox of simple primitives for compilers to use rather than prescribe specific models.

CHERI capabilities can also protect the stack. For example, stack pointers can be cast to capabilities to take advantage of bounds checking. We have an experimental version of the compiler that protects individual frames by using stack capabilities to eliminate stack overflow.

Pointer subtraction is not supported natively by CHERI capabilities because they do not have an internal index. An external integer register index can be used if pointer subtraction is required. Capabilities can be indexed with signed register and immediate values, but an access will throw an exception if the final offset is out of range.

The CHERI ABI defines eight capability-argument registers for passing capabilities in a function call, which greatly alleviate register pressure when replacing a software fat-pointer scheme, which would otherwise require at least two general-purpose registers for each pointer. However, stack spills will still have a larger cache footprint due to increased register size. Inter-domain calls (which incur a larger overhead to support mutual distrust) are not yet integrated into our compiler.

We might note that consistency between base and bounds can be a problem even for hardware fat pointer implementations, as we describe in Section 6.4. However, CHERI capabilities used as fat pointers avoid race conditions by updating capability fields and tags atomically.

5.2. Managed language runtime support

Languages that do provide memory safety must enforce it with the available instructions. CHERI capabilities provide a level of object support in the instruction set to allow managed language runtimes and the JIT compilers that target them to be simpler, faster, and more secure. The segments-as-objects model was one of the basic motivations for iAPχ432 segmentation [30], and in turn for Intel 80286 segments [7], but these segment-table approaches do not scale to arbitrary program size or data complexity. In contrast, a straightforward application of capability registers as pointers and of protected calls (see Section 11) on invocations would provide scalable, strong hardware protection of language objects.

5.3. Sandboxing and compartmentalization

Memory protection constrains the effects of code, which can be used not just for improvements in robustness, but also for security. The CHERI model conveniently and efficiently supports application sandboxing and compartmentalization. Conventional binaries are sandboxed in micro-address spaces within existing processes by constraining **C0** and **PCC**. CHERI-aware programs use capabilities to describe more granular compartmentalization in which memory and object rights are safely delegated between multiple protection domains within a UNIX process.

6. Functional comparison of protection models

In this section, we review several proposed protection models supporting safety and compartmentalization, and compare them with CHERI. Table 2 considers these models in terms of our protection criteria: unprivileged use; fine-grained control; unforgeable references; read, write, and execute permissions; pointer safety, segment scalability; domain scalability; and incremental deployability to current programming languages.

CHERI is able to provide the stronger memory-protection and domain-transition properties of the M-Machine capability model with greater adoptability due to inclusion of a general-purpose MMU and a complete fat-pointer representation. However, this comes with increased memory traffic relative to M-Machine, more invasive instruction set additions relative to Hardbound, and reduced binary compatibility relative to iMPX. Protection, representation, and performance tradeoffs are considered in greater detail in Sections 7 and 8.

Protection mechanism	Unprivileged use	Fine-grained	Unforgeable*	Access control	Pointer safety	Segment scalability	Domain scalability	Incremental deployment
MMU	-	-	-	✓	-	-	-	✓
Mondrian	-	✓**	-	✓	-	✓	-	✓
Hardbound	✓	✓	✓	-	✓	✓	n/a	✓
iMPX	✓	✓	✓	-	✓	✓	n/a	✓
iMPX Fat Pointers	✓	✓	-	-	✓	✓	n/a	-
M-Machine	✓	-	✓	✓	✓	✓	✓	-
CHERI	✓	✓	✓	✓	✓	✓	✓	✓

*Unforgeability in the context of protection-domain-free models refers to the difficulty of constructing an unauthorized pointer to an object.
**Mondrian supports fine-grained heap protection, but not fine-grained stack or global protection.

Table 2: Comparison of address-validity, pointer-validity (table-based), and pointer-validity (fat-pointer based) models

6.1. Conventional Memory Management Units (MMUs)

Traditional Memory Management Units (MMUs) map a (possibly sparse) virtual address space into physical memory via a *page table*, with permissions assigned to each page. Kernels used MMUs to isolate processes from one another (implementing protection domains), with shared memory supported by mapping the same physical page into multiple address spaces.

The MMU approach fails most of our requirements for in-address space protection, but is the only widely deployed protection mechanism today. Page-based protection is coarse-grained: most implementations have a minimum page size of 4KB limiting bounds-checking precision. Coarse-grained MMU protection can detect either overflows or underflows for small objects, but not both; single-object allocations will waste physical memory, and *guard pages* between allocations will consume virtual address space. MMUs implement only *address validation*, not *pointer safety*: a sufficiently large overflow may miss the guard page and write to another allocation.

MMU isolation can also be used for *application compartmentalization*, such as in the Chromium web browser where tabs run in separate processes to improve robustness and security. MMU process isolation proves expensive due to TLB capacity limits that are exacerbated by aliasing between shared pages. As a result, applications limit use of sandboxes, thus reducing isolation in favor of performance [33].

CHERI inherits all the benefits of the MMU and adds capability protection for principled fine-grained protection within an address space. A key virtue of the MMU as an *address validation* approach is centralized management, which simplifies address-space revocation, a classic weakness of capability machines. While CHERI does not accelerate revocation of pointers that use the capability mechanism, the operating system can manipulate mappings of the underlying pages to enforce revocation. To facilitate this, CHERI extends page table entries with bits to to authorize capability loads and stores. This also allows the OS to implement shared memory between processes that cannot act as a channel for passing capabilities.

6.2. Mondrian memory protection

Mondrian memory protection is a model that allows fine-grained memory protection to be layered on top of page-based virtual memory, to facilitate multiple protection domains [45]. The conventional page table is supplemented by word-granularity in-memory protection tables that contain permissions managed by the supervisor. These tables populate a Protection Look-aside Buffer (PLB) analogous to the TLB in the MMU. In addition, a set of *sidecar registers* are paired with general-purpose registers to reduce PLB pressure. No userspace ISA changes are required to support Mondrian, which enhances incremental deployment.

Mondrian relies on supervisor mode to maintain its protection table, and thus incurs a domain switch for each allocation and free event. At the time Mondrian was evaluated, such domain switches would have been common already due to invocations of `sbrk()` or `mmap()` system calls; however, userspace allocators now aggressively cache memory in order to amortize domain switches and lock contention. Reintroducing domain switches for Mondrian would significantly impair segmentation scalability. In contrast, CHERI does not require system calls to create new segments.

As with an MMU, Mondrian provides address validation rather than pointer validation (albeit at much finer granularity). Thus, all allocations must be padded to introduce guard regions, which raises similar concerns to MMU guard pages. Smaller pads are possible than with pages, while reducing the threshold at which overflows can be detected. This prevents Mondrian from providing effective protection for sub-allocations such as array entries or individual stack frames, and limits its usefulness for fine-grained protection. This is particularly a concern today when many classes of exploitable security vulnerabilities are premised on overflows with attacker control over inputs to arithmetic. At somewhat greater cost to ISA-level compatibility, CHERI provides pointer safety suitable for bounds checking on densely packed (and divisible) memory locations.

Protection-domain scalability in Mondrian is limited: each domain requires its own complete protection table, each with substantial memory and initialization expense. In CHERI, userspace protection does not involve tables. Instead, protection information is embedded in pointers, and protection domains are infinitely scalable and defined only by the set of capability pointers reachable by the current thread.

6.3. Hardbound

Hardbound [12] is a hardware-assisted fat-pointer model grounded in software bounds-checking research. Unlike the MMU and Mondrian, Hardbound provides pointer safety, not just address validation, and is able to enforce sub-allocations and stack variables. Hardbound maintains a shadow table of *base* and *bounds* values for each pointer-aligned virtual memory location, and another table of *tag* bits to identify pointers. Memory bounds are initialized by a modified memory allocator for heap objects, and ideally also by a modified compiler for stack and global objects. The Hardbound simulated processor propagates bounds into the shadow table and via registers, and verifies bounds when pointers are dereferenced. Libraries and applications that are not recompiled to assign bounds to pointers will experience less mitigation. Interestingly, Hardbound does not provide permission bits necessary to enforce certain references such as `const` read-only.

Hardbound, the M-Machine, and CHERI rely on tags to robustly distinguish pointers from other data in memory. However, Hardbound fat pointers are forgeable: the `setbound` instruction allows arbitrary bounds, and the tables are accessible via virtual memory. As a result, unlike the M-Machine and CHERI, Hardbound pointers do not constitute a protection domain. Hardbound is also a CISC design that proposes a microcode implementation, and requires transactional memory to write to three table entries atomically.

As with Mondrian and CHERI, Hardbound is concerned with incremental adoption. Hardbound executables can run on legacy hardware by careful use of NOP opcodes, and ABIs are maintained by retaining native pointer size.

Hardbound performance is limited by TLB overhead (due to sparse table access), and by memory overhead (due to inflated pointers). For example, a single memory instruction in Hardbound might experience two additional TLB misses for the tag and bound tables, compared to the same access in CHERI. To mitigate these, Hardbound relies on fat-pointer compression, and wherever possible stores bounds within the tag or in the user pointer. We observe that unused pointer bits are not always available, because some language runtimes (such as Java, JavaScript, and Objective-C) use these bits for their own optimizations.

6.4. iMPX

Intel's recently announced Memory Protection Extensions (iMPX) [17] describe additions to the x86 ISA to provide hardware-assisted bounds checking. At present, Intel has not shipped a hardware implementation of iMPX, although a simulator and compiler extensions are available. As with Hardbound, bounds information can be stored in a table within a process's virtual address space. There are several important differences, however:

- Bounds are not automatically propagated, but are explicitly loaded and stored into a new register set by instructions.
- Bounds can originate in architecturally supported shadow tables, but also in software-defined locations (e.g., stored adjacent to the pointer itself).
- Rather than explicit tags in a second table, iMPX includes the original pointer value in the bounds-table entry. If the bounds are loaded against a pointer of a different value, the bounds will be ignored. This preserves compatibility with legacy code which may not update bounds.
- Bounds checking is performed using explicit instructions.
- There is no support for pointer compression. Each 64-bit pointer consumes 320 bits: the original pointer along with 256 bits of metadata including base, bounds, the expected pointer value, and 64 reserved bits.
- iMPX uses a hierarchical protection table.

Without Hardbound's pointer compression, iMPX experiences significant memory overheads, even compared to 256-bit CHERI capabilities. However, support for storing bounds in arbitrary addresses enables a traditional, consecutive fat-pointer layout, which has greater locality. As with Hardbound, iMPX does not support permission bits, although reserved space in the shadow table might be used for this in the future.

iMPX strongly emphasizes compatibility. As with Hardbound, iMPX instructions decode as NOPs in older ISAs, and will be ignored by older processors. Its ABI will reset bounds registers for iMPX-unaware code for compatibility. iMPX's table mode has greater binary compatibility than CHERI as pointer sizes remain the same – but hinders safety and efficiency. As with Hardbound, portions of the pointer may be stored in different cache lines, and require transactional memory to preserve atomicity. If atomicity is not maintained, iMPX fails open. If multiple threads interfere such that pointer and table entries become inconsistent, *any* memory accesses for the pointer are silently allowed.

6.5. M-Machine

The M-Machine [5] is a 64-bit tagged-memory capability system design using *guarded pointers* to implement fine-grained memory protection for pointer safety. M-Machine pointers are unforgeable. They define a protection domain within a single address space, and protection-domain switching is supported. The M-Machine implementation depends on a supervisor mode for pointer creation and manipulation, though the authors propose guarded user-mode instructions. CHERI differs from the M-Machine in the following ways:

- The M-Machine uses an MMU only for paging support, whereas CHERI uses an MMU to support multiple address spaces (processes), and uses capabilities for protection and

domain switching within each address space.

- The M-Machine provides a pointer compression scheme that reduces fat pointers to 64 bits at the cost of granularity: only power-of-two aligned and sized segments are supported. This reduces memory and cache footprint, at the cost of granularity and adoptability; padding is required for common structures that break binary layouts.
- CHERI provides explicit pointer/capability conversions and sandboxing to support (and confine) legacy code.
- CHERI allows capability manipulation in usermode.

The M-Machine is an efficient capability-based addressing implementation, but almost entirely sacrifices compatibility with current software designs. CHERI adopts similar design elements, but with stronger focus on compatibility, adoptability, and efficient compiler manipulation of capabilities.

7. Limit study

Protection schemes vary widely in functionality and performance. To understand performance tradeoffs, we performed a simulation-based limit study on pointer-intensive benchmarks. The study measured instruction rate, memory traffic overhead, system-call rate, and memory storage overhead (Figure 3). We compared 256-bit CHERI with six models: the Mondrian model; two variations on iMPX (ABI-preserving look-aside tables, compiler-managed fat pointers); software fat pointers; Hardbound; and the M-Machine. We also added a 128-bit CHERI variant: 256-bit capabilities allow flexibility in experimentation, but we would expect a production implementation to use a smaller size. We adapted each model to 64-bit MIPS:

Mondrian We extend Mondrian to a 40-bit virtual address space, and simulate its vector-table model with indices to the first- and mid-level tables stretched to 14 bits. Records are extended to 64 bits and hold permissions for 16 nodes rather than 8, giving finer granularity. We assume a hardware read of the table but simulate a software table fill based on a minimal table fill algorithm in C.

iMPX Fat-pointer and table-based models translate directly.

Soft fat pointers Fat-pointer loads, stores, and bounds checks use general-purpose instructions.

Hardbound We extend base and bounds information to 64 bits. We retain a direct offset for the bounds table and model a 128-bit table access for every load (or store) of an incompressible pointer. Compressed pointers encode up to 1024 bytes of length in 8 unused bits in the pointer and require length to be 4-byte word aligned. We model a 2-bit tag for each 64-bit word stored in a separate table in memory.

We used the Olden benchmarks [34], a suite developed for distributed shared-memory research that has become popular in bounds-checking research due to its focus on pointer-based data structures. The benchmarks use a range of data structures, memory footprints, and workloads to exercise various pointer access patterns and densities.

To measure performance for each approach, we recorded complete instruction traces of Olden benchmarks on our

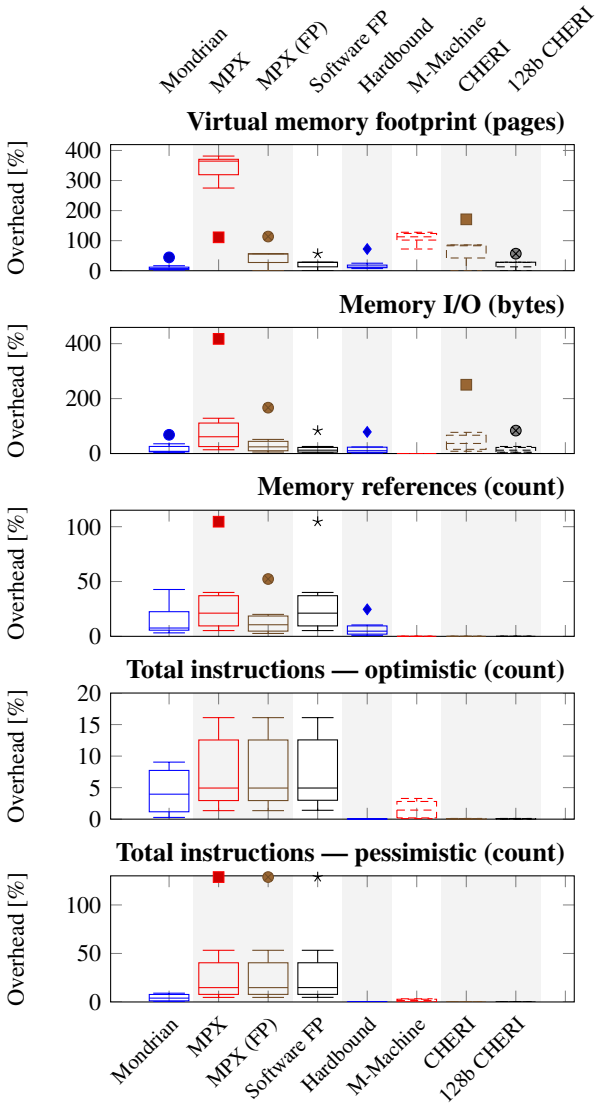

Figure 3: Simulated overheads of Olden benchmarks

baseline MIPS implementation in hardware. We then extracted information relevant to bounds checking: C memory-management functions such as `malloc()` and `free()`, and all memory loads and stores. This allowed us to track accesses to all objects in memory-mapped segments (globals), heap and stack. We simulated extra memory accesses, instructions, TLB and cache behavior, and system calls that would result from ideal implementations of each model. Performance results are presented as normalized overhead against the baseline.

The number of virtual memory pages touched by the process reflects both TLB and cache pressure. Caches optimize for locality of reference so cache performance should degrade as the range of pages increases, even if the total traffic remains constant. The iMPX case has the highest page overhead. The iMPX table contains more than 4 pages for each page of memory containing pointers, maintaining 256 bits in the leaf nodes for each 64-bit memory location. The Hardbound model

uses a simpler table structure and 128 bits of metadata per pointer, but must also maintain a tag table. Interestingly, the M-Machine performs poorly by the page metric due to padding allocations to powers of two. CHERI and the other simple fat-pointer approaches have comparatively small page overhead, as the additional data is packed into existing data and the larger structures will only sometimes spill onto another page. These will use more cache, but not significantly more TLB entries.

The simplest memory metric is the total number of bytes read or written. As expected, the table walk in iMPX requires significantly more memory accesses than any other scheme. CHERI generates more traffic on these pointer-heavy benchmarks due to its larger pointer size; however, the proposed 128-bit variant is competitive with most of the other models. Mondrian uses the smallest amount of memory traffic, as it does not provide per-pointer bounds.

The number of individual loads and stores is presented separately from the number of bytes stored, as many structures in the CPU scale with the number of independent transactions rather than with the total size of transactions. CHERI, Hardbound, and the M-Machine all do well on this metric. In the case of CHERI and the M-Machine, where metadata is stored inline, they have negligible increases in this category. Hardbound does well because it succeeds in compressing a large proportion of the pointers. Mondrian has a similarly low overhead because it does not associate protection with pointers and therefore does not cache larger regions of protection.

Perhaps the simplest form of overhead is increased instruction count; this is not an accurate predictor of performance in superscalar implementations, but does impact instruction cache usage. We show optimistic and pessimistic instruction count overheads. The optimistic model assumes bounds can be checked once on every pointer load; the pessimistic model assumes that bounds must be checked on every pointer dereference. These are identical for hardware fat-pointer approaches, where all dereferences are checked implicitly and the only additional instructions are to set the bounds on new allocations. CHERI and Hardbound require a single instruction; Mondrian requires a system call to modify the protection table. Explicit bounds loads and checks in iMPX and the software fat-pointer approaches have the most overhead.

The CHERI model proves competitive in all major metrics (especially in its 128-bit variant), indicating that there are no major limitations to our approach.

8. Performance measurement

In addition to our limit study, we also measured the performance of four of the Olden benchmarks running on CHERI implemented in FPGA. We compiled each benchmark with conventional MIPS code generation, MIPS code with automatically generated bounds checks inserted by CCured [28], and CHERI memory safety driven by manual insertion of `__capability` annotations. Each was compiled with our modified version of LLVM, and run as a userspace program on FreeBSD 10 with CHERI extensions.

CCured was chosen as a best-of-breed software bounds-checking implementation as it elides bounds checks where it can statically prove them unnecessary. CHERI will always enforce bounds dynamically in hardware. In contrast to CHERI, the CCured version is not thread-safe due to non-atomic pointer access (as noted in its documentation).

The four benchmarks were `bisort`, `mst`, `treeadd` and `perimeter`. To enable comparison, we ran the benchmarks with the same parameters as used in the evaluation of Hardbound: `bisort 250000 0`, `mst 1024 0`, `treeadd 21 1 0` and `perimeter 12 0`. Figure 4 shows execution-time overhead relative to the unsafe MIPS baseline, with total time decomposed into allocation and computation phases.

In the `bisort` benchmark, CHERI experiences a very small overhead while allocating memory for each node in the tree. CHERI requires one extra instruction for each allocation to set bounds, while the software-enforcement case is significantly more complex. The sorting phase involves traversing the tree and swapping pointers. This phase is dominated by cache miss time. Unsafe nodes are 24-bytes, which fit more efficiently in our 32-byte cache lines than CHERI's 96-byte nodes. Due to the similar data structure used, `treeadd` has comparable performance profile to `bisort`.

The `mst` benchmark shows a different access pattern with two contiguous allocations to build the graph and a linear read to compute the minimum spanning tree. When building the graph, the total run time is dominated by the hash calculations that are the same in both cases. When computing the minimum spanning tree, as with `bisort`, the dominant factor is cache misses. CHERI experiences more cache misses because of large capabilities, though this linear case would be alleviated with cache prefetching.

The `perimeter` benchmark shows a small performance improvement during the memory allocation phase when capabilities are used. This is probably because the slowdown due to capabilities is smaller than the performance variation due to other effects, such as allocation alignment in `malloc`.

CHERI outperforms CCured substantially in all configurations. In `bisort`, `treeadd`, and `perimeter`, this is due to improvements in both allocation and computation. In `mst`, CCured is effective in eliding inner-loop bounds checks during computation, achieving greater cache efficiency by avoiding fat-pointer overhead. Similar elision could also be applied to CHERI to selectively utilize capabilities.

Figure 5 shows the percentage slowdown of CHERI relative to MIPS code for increasing data-set sizes. For very small sets, overhead is negligible. As working set-size increases, capability cache pressure grows faster than for unprotected code. This leads to visible 'steps' as the 16KB L1 cache, 64KB L2 cache, and TLB covering 1MB overflow.

Our benchmarks are consistent with those predicted by the limit study: MIPS and CHERI execution times remain very close when data is cached, but performance degrades where

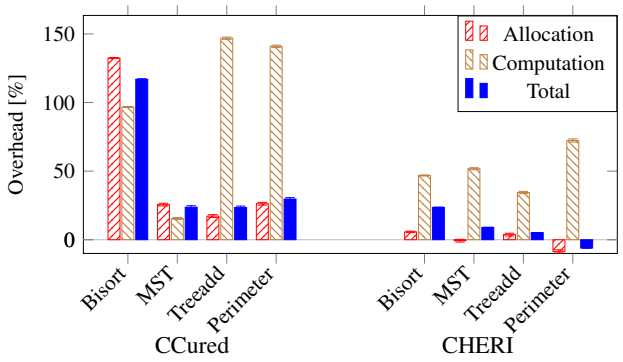

Figure 4: Benchmark results comparing unmodified MIPS code to software and hardware enforcement

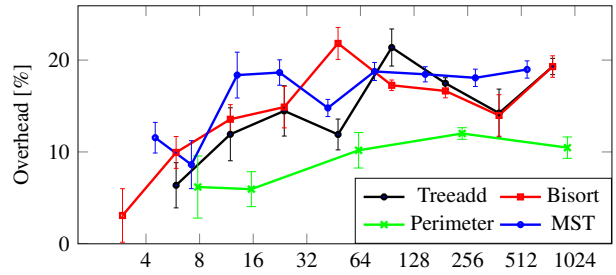

Figure 5: Slowdown for CHERI at different heap sizes (KB)

pointer-size is dominant. These results reconfirm that CHERI will benefit from capability compression, and perhaps also elision techniques, although performance is acceptable even in pointer-heavy benchmarks.

9. Area and speed cost

CHERI capability support adds both area and speed cost to our FPGA soft core. A synthesis of CHERI, excluding peripherals, consumes 32% more logic elements than BERI. This overhead includes not only the capability coprocessor and the tag manager, but also logic in the main pipeline to allow loading and storing 256-bit capabilities into the data cache. Figure 6 shows a synthesis of CHERI broken into components to indicate area cost by module. A commercial implementation would optimize capability size and may have a wide vector path to memory that might also be used for the capabilities.

Capability support also affects the timing of our implementation due to wider paths and increased complexity in the pipeline; our current implementation reduces clock speed by 8.1%, as BERI achieves a maximum frequency of 110.84 MHz, while the capability coprocessor reaches 102.54 MHz. Little attempt has been made to improve the timing of either case beyond 100MHz. An implementation that expects less-frequent capability updates could decouple capability operations from the main pipeline at a performance cost.

10. Limitations

The CHERI approach is not without limitations, many of which we hope to address in future work. Perhaps most importantly, CHERI's protection features require adapting and recompiling application code. Despite capability-qualified pointers supporting all operations permitted by the C standard with the exception of subtraction, practical C implementations tolerate undefined pointer behaviors that CHERI capabilities will not. The Olden suite, for example, was trivially adapted to CHERI by simply tagging pointers with `__capability`. Some applications, however, routinely construct pointers that extend significantly beyond the end of valid buffers (disallowed by the C specification), which will trigger exceptions on CHERI. In performing a more complex adaptation of `tcpdump` to use CHERI capabilities, we encountered many such uses — some of which led to undesirable (and potentially exploitable) out-of-bounds memory accesses with conventional compilation.

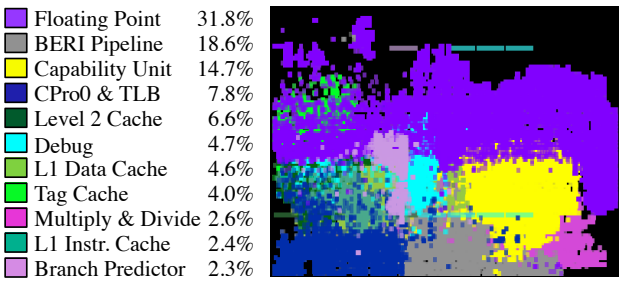

Figure 6: CHERI layout on FPGA

Unadapted MIPS code with ambient authority in the same address space as CHERI-adapted code leaves adapted code vulnerable to buggy or exploited MIPS code. This can be mitigated by deploying explicit sandboxing of unadapted code, at some cost to complexity and performance. In future work, we hope to explore techniques for automated adaptation of application code and for sandboxing of unmodified code.

11. Future work

Our larger research effort has now just begun. We intend to explore compiler and system ramifications of the current hardware extensions, and also to explore the utility of further extensions. We expect to be able to demonstrate efficient *sandboxing* using CHERI protection primitives, and explore software models and scalability.

We are experimenting with several mechanisms for *protected domain crossing*. Our current prototype traps to the OS to emulate a protected procedure-call instruction, but we intend to provide a hardware (or hardware-assisted) implementation as the software model matures.

Our current work provides *spatial* memory safety. The presence of tagged memory also provides opportunities to enforce *temporal* safety. Tags allow us to identify all references, so we can provide accurate garbage collection to low-level languages such as C. Possibilities include a non-reuse allocator (to eliminate most dangling pointer errors) that periodically runs a tracing pass to identify reusable address space.

Distinguishing integers and pointers in the memory hierarchy also enables cache hinting for pre-fetching or coherency.

We are also investigating the use of the capability mechanism for interactions between coprocessors (e.g., GPUs) and userspace code, without requiring operating-system mediation.

12. Related work

Strong memory protection has long been sought for computer systems [10, 35]. Initially, segmentation (and capabilities) were developed to protect and relocate program structures [11, 23]. Fixed-size paging, however, eventually became dominant to allow operating systems to efficiently manage multiple programs on the same machine [14, 32]. For example, Intel implemented fine-grained segmentation in the 80286 but it was eventually dropped in x86-64 [8]. Nevertheless, computers that optimized for protecting individual programs often support segmentation, for example the Burroughs large systems [25, 43], and the modern ARM Cortex R cores [15].

For systems without hardware segmentation in user space, programmers turned to statically enforced memory protection [6] and/or run-time checks [9]. Managed-language virtual machines automate these mechanisms to enforce an object memory model [16] and incur significant overhead as a result [31]. Other attempts have been made to provide bounds checking in software for C [2, 37, 38], but with prohibitive overhead for a performance-oriented language. To achieve coarse-grained software sandboxing of program components, efforts have been made to use process separation [36, 40], disjoint TLB sets in a shared process, static analysis of specially compiled binaries [24, 39, 48], and delegation of hypervisor support to userspace [3].

Another approach is memory protection in user-space code for managed languages [44, 47] that implements certain aspects of the virtual machine's memory model in the hardware. Such approaches are less applicable when computers run software written in multiple languages, with different memory models. Commercial approaches, such as ARM's Jazelle and Thumb-2EE, have provided bounds-checking instructions aimed at efficiently implementing languages such as Java without a specialized Java processor.

13. Availability

The CHERI ISA reference is available as a technical report [41]. In the interests of experimental reproducibility, we have open sourced our BERI prototype and software stack:

`http://www.bericpu.org/`

`http://www.chericpu.org/`

We hope that this will not only encourage experimentation with capability-based techniques such as CHERI, but also support further combined hardware-software research.

14. Conclusions

Fine-grained memory protection is vital for increasing security and robustness in contemporary software. To date, only coarse-grained MMU-based protection models have had wide impact on computer systems, despite many historic proposals for capability systems and more recent proposals for protection tables and fat pointers. We have hybridized a capability model with an MMU-based design to demonstrate that a tagged capability system can provide efficient fine-grained protection as well as compatibility with current source-code and binary corpora. Incremental adoption is critical, as we live in a world with large established software codebases. Mandatory rewriting – or even recompiling – is no longer acceptable for deployment.

Earlier capability systems have seen limited adoption, although they largely predated widespread Internet connectivity and security threats. More recently, processor vendors have introduced new security features (i.e., TrustZone [1] and iMPX [17]), which might suggest that there could soon be a sufficient market for capability-based addressing. Hybrid software capability systems (e.g., Capsicum [40]) have allowed capabilities (also neglected in the software community) to be adopted on a larger scale, due to improved legacy-code compatibility. Similarly, a key contribution of the CHERI approach is adoptability that is incremental, rather than ground-up.

Our feature comparison and limit study illustrate how protection-model design choices made by several published schemes can trade off among protection, performance, and compatibility. They demonstrate that CHERI performs competitively with all schemes while often providing stronger protection, and that capability-size optimizations would further improve CHERI performance. We also demonstrate that incremental deployment is possible and that parts of a program can benefit from complete spatial safety – without having to rewrite or recompile entire codebases.

CHERI exemplifies the RISC philosophy of creating simple instructions that are designed to be useful to compilers. As a result, implementation complexity is sufficiently low for consideration in modern processors, and performance is noticeably faster than weaker enforcement in software.

14.1. Acknowledgments

We thank our colleagues – especially Ross Anderson, Gregory Chadwick, Nirav Dave, Khilan Gudka, Wojciech Koszek, A Theodore Markettos, Ed Maste, Andrew W. Moore, Will Morland, Prashanth Mundkur, Steven J. Murdoch, Philip Paeps, Colin Rothwell, Hassen Saidi, Stacey Son, and Bjoern Zeeb; we also thank our anonymous reviewers for their feedback.

This work is part of the CTSRD Project that is 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 paper 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. We gratefully acknowledge Google, Inc. for its sponsorship.

References

[1] T. Alves and D. Felton, "ARM TrustZone: Integrated hardware and software security," *Information Quarterly*, vol. 3, no. 4, July 2004.

[2] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, ser. PLDI '94. New York, NY, USA: ACM, 1994, pp. 290–301.

[3] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: safe user-level access to privileged CPU features," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, ser. OSDI'12, 2012, pp. 335–348.

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

[5] N. P. Carter, S. W. Keckler, and W. J. Dally, "Hardware support for fast capability-based addressing," *SIGPLAN Not.*, vol. 29, no. 11, pp. 319–327, Nov. 1994.

[6] B. Chess, "Improving computer security using extended static checking," in *Proceedings of the 2002 Symposium on Security and Privacy*. Oakland, California: IEEE Computer Society, May 2002, pp. 160–173.

[7] R. Childs Jr, J. Crawford, D. House, and R. Noyce, "A Processor Family for Personal Computers," *Proceedings of the IEEE*, vol. 72, no. 3, pp. 363–376, 1984.

[8] S. Cleveland, "x86-64 technology white paper," Advanced Micro Devices, Tech. Rep., 02 2002.

[9] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, vol. 2. IEEE, 2000, pp. 119–129.

[10] P. Denning, "Virtual memory," *ACM Computing Surveys (CSUR)*, vol. 2, no. 3, pp. 153–189, 1970.

[11] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Commun. ACM*, vol. 9, no. 3, pp. 143–155, 1966.

[12] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: architectural support for spatial safety of the C programming language," *SIGARCH Comput. Archit. News*, vol. 36, no. 1, pp. 103–114, Mar. 2008.

[13] J. Evans, "A scalable concurrent malloc(3) implementation for FreeBSD," in *BSDCan*, 2006.

[14] J. Fotheringham, "Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store," *Communications of the ACM*, vol. 4, no. 10, pp. 435–436, 1961.

[15] A. Frame and C. Turner, "Introducing new ARM Cortex-R technology for safe and reliable systems," ARM, Tech. Rep., 03 2011.

[16] J. Gosling and H. McGilton, *The Java language environment*. Sun Microsystems Computer Company, 1995, vol. 2550.

[17] Intel Plc., "Introduction to Intel® memory protection extensions," http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions, July 2013.

[18] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *ATEC '02: Proceedings of the USENIX Annual Technical Conference*, 2002, pp. 275–288.

[19] 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*, vol. 53, pp. 107–115, June 2009.

[20] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, Jr., and A. DeHon, "Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *20th ACM Conference on Computer and Communications Security*, November 2013.

[21] L. Lam and T. Chiueh, "Checking array bound violation using segmentation hardware," in *IEEE International Conference on Dependable Systems and Networks*, 2005, pp. 388–397.

[22] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and runtime optimization*, ser. CGO '04, 2004, pp. 75–86.

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

[24] 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.

[25] A. J. Mayer, "The architecture of the Burroughs B5000: 20 years later and still ahead of the times?" *ACM SIGARCH Computer Architecture News*, vol. 10, no. 4, pp. 3–10, 1982.

[26] M. K. McKusick and G. V. Neville-Neil, *The design and implementation of the FreeBSD operating system*. Pearson Education, 2004.

[27] J. Navarro, S. Iyer, P. Druschel, and A. L. Cox, "Practical, transparent operating system support for superpages," in *OSDI*, 2002.

[28] G. C. Necula, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy code," in *ACM SIGPLAN Notices*, vol. 37, no. 1, 2002, pp. 128–139.

[29] D. A. Patterson and C. H. Sequin, "RISC I: A reduced instruction set VLSI computer," in *Proceedings of the 8th Annual Symposium on Computer Architecture*, 1981, pp. 443–457.

[30] F. J. Pollack, G. W. Cox, D. W. Hammerstrom, K. C. Kahn, K. K. Lai, and J. R. Rattner, "Supporting Ada memory management in the iAPX-432," in *ACM SIGARCH Computer Architecture News*, vol. 10, no. 2, 1982, pp. 117–131.

[31] F. Qian, L. Hendren, and C. Verbrugge, "A comprehensive approach to array bounds check elimination for Java," in *Compiler Construction*. Springer, 2002, pp. 325–341.

[32] B. Randell and C. Kuehner, "Dynamic Storage Allocation Systems," *Communications of the ACM*, vol. 11, no. 5, pp. 297–306, 1968.

[33] 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*. New York, NY, USA: ACM, 2009, pp. 219–232.

[34] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren, "Supporting dynamic data structures on distributed-memory machines," *ACM Trans. Program. Lang. Syst.*, vol. 17, no. 2, pp. 233–263, Mar. 1995.

[35] J. Saltzer and M. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, September 1975.

[36] M. Schroeder and J. Saltzer, "A hardware architecture for implementing protection rings," *Communications of the ACM*, vol. 15, no. 3, March 1972.

[37] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: A fast address sanity checker," in *USENIX ATC*, vol. 12, 2012.

[38] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic, "Heapmon: a helper-thread approach to programmable, automatic, and low-overhead memory bug detection," *IBM J. Res. Dev.*, vol. 50, no. 2/3, pp. 261–275, Mar. 2006.

[39] 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*, New York, NY, USA, 1993, pp. 203–216.

[40] 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.

[41] R. N. M. Watson, P. G. Neumann, J. Woodruff, J. Anderson, D. Chisnall, B. Davis, B. Laurie, S. W. Moore, S. J. Murdoch, and M. Roe, "Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture," University of Cambridge, Computer Lab., Tech. Rep. UCAM-CL-TR-850, May 2014. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-850.html

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

[43] A. Wilkinson et al., "A penetration study of a Burroughs large system," *ACM Operating Systems Review*, vol. 15, no. 1, pp. 14–25, January 1981.

[44] I. Williams and M. Wolczko, "An object-based memory architecture," in *Fourth International Workshop on Persistent Objects*. Morgan Kaufmann, 1990, pp. 114–130.

[45] E. Witchel, J. Cates, and K. Asanović, *Mondrian memory protection*. ACM, 2002, vol. 37, no. 10.

[46] E. Witchel, J. Rhee, and K. Asanović, "Mondrix: Memory isolation for Linux using Mondriaan memory protection," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

[47] G. Wright, M. L. Seidl, and M. Wolczko, "An object-aware memory architecture," *Science of Computer Programming*, vol. 62, pp. 145–163, 2006.

[48] 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 *SP '09: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, 2009, pp. 79–93.