CHERI Concentrate: Practical Compressed Capabilities

Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert Norton, Thomas Bauereiss, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W. Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, Simon W. Moore

Abstract—We present CHERI Concentrate, a new fat-pointer compression scheme applied to CHERI, the most developed capability-pointer system at present. Capability fat pointers are a primary candidate to enforce fine-grained and non-bypassable security properties in future computer systems, although increased pointer size can severely affect performance. Thus, several proposals for capability compression have been suggested elsewhere that do not support legacy instruction sets, ignore features critical to the existing software base, and also introduce design inefficiencies to RISC-style processor pipelines. CHERI Concentrate improves on the state-of-the-art region-encoding efficiency, solves important pipeline problems, and eases semantic restrictions of compressed encoding, allowing it to protect a full legacy software stack. We present the first quantitative analysis of compiled capability code, which we use to guide the design of the encoding format. We analyze and extend logic from the open-source CHERI prototype processor design on FPGA to demonstrate encoding efficiency, minimize delay of pointer arithmetic, and eliminate additional load-to-use delay. To verify correctness of our proposed high-performance logic, we present a HOL4 machine-checked proof of the decode and pointer-modify operations. Finally, we measure a 50% to 75% reduction in L2 misses for many compiled C-language benchmarks running under a commodity operating system using compressed 128-bit and 64-bit formats, demonstrating both compatibility with and increased performance over the uncompressed, 256-bit format.

Index Terms—Capabilities, Fat Pointers, Compression, Memory Safety, Computer Architecture

1 Introduction

Intel Memory Protection Extensions (MPX) and Software Guard Extensions (SGX), as well as Oracle Silicon Secured Memory (SSM), signal an unprecedented industrial willingness to implement hardware mechanisms for memory safety and security. As industry looks to the next generation, capability pointers have become a primary candidate to conclusively solve memory safety problems. Capability pointers are stronger than fault detection schemes such as MPX and SSM, and are able to achieve provable containment at the granularity of program-defined objects that is as strong as address-space separation.

The greatest cost for capability pointers involves the object bounds encoded with each pointer to enforce memory safety. Encoding both upper and lower bounds as well as a pointer address requires either larger capabilities [1] or restrictions on region properties, semantics, and address space [2], [3].

This paper presents CHERI Concentrate (CC), a compression scheme applied to CHERI, the most developed capability-pointer system at present. CC achieves the best published region encoding efficiency, solves important pipeline problems caused by a decompressed register file, and eases semantic restrictions due to the compressed encoding. The contributions of this paper are:

- A floating-point bounds encoding with an *Internal Exponent* that provides maximum precision for small objects, spending bits to encode an exponent only for larger and less common objects.
- The first quantitative characterization of capability operations in compiled programs to inform capability instruction optimization.
- A power-of-two *Representable Region* beyond object bounds to allow temporarily out-of-bounds pointers, enabling compatibility with a broad legacy code base.
- A *Representability Check* for pointer arithmetic with delay comparable to a pointer add, enabling integration with standard processor designs.

CC improves efficiency over Low-Fat Pointers, the previous best capability bounds format, by inferring the most significant bit of the Top field and by encoding the exponent within the bounds. CC also improves both semantics and timing by allowiny out-of-bounds pointer manipulations, which simplifies the pointer arithmatic check allowing it to be performed directly on the compressed format.

- *Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert Norton, Thomas Bauereiss, David Chisnall, Khilan Gudka, Nathaniel Filardo, Theo Markettos, Michael Roe, Robert Watson, Simon Moore are with the Department of Computer Science and Technology, University of Cambridge, England. Email is {firstname.lastname}@cl.cam.ac.uk.*
- *Brooks Davis and Peter Neumann are with SRI International. Email is {firstname.lastname}@sri.com.*

This work is part of the CTSRD, ECATS, and CIFV projects sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237, HR0011-18-C-0016, and FA8650-18-C-7809. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government. Approved for Public Release, Distribution Unlimited. We also acknowledge the EPSRC REMS Programme Grant [EP/K008528/1], the EPSRC Impact Acceleration Account [EP/K503757/1], Arm Limited, and Google, Inc.

2 Background

The importance of architectural support for fine-grained memory protection has been demonstrated in research (e.g., Mondriaan Memory Protection [4], [5]; Hardbound [6]) and in industry (e.g., MPX [7]). In particular, Mondriaan pointed out that current operating systems use paged memory for both *protection* and *virtualization*, creating tension between granularity and performance. Capability pointer systems use bounded pointers for fine-grained protection, and use paged memory only for virtualization.

Early capability-pointer machines include the CAP computer [8], Intel iAPχ432 [9], and the Intel i960 [10]. These machines used indirection to efficiently store object bounds in an object table. In contrast, more recent capability machines, including the M-Machine [2] and the CHERI processor [1], [11], [12], [13] encode object bounds directly in unforgeable *fat pointers*, avoiding an additional memory access to an object table, or an associative lookup in an object cache.

The increased memory footprint due to encoding bounds in every pointer can be a major challenge for fat-pointer capability schemes. Fat-pointer compression techniques, used by the M-Machine [2], Aries [14], and Low-Fat Pointers [3], exploited redundancy between a pointer address and its bounds to reduce their storage requirements. We learn from these techniques, improve upon them, and solve additional challenges to apply them in the context of a conventional RISC pipeline and a large legacy software base.

Notes on notation: In this paper we use the notation $U'V$ to indicate a field named U composed of V bits. $U[W:X]$ indicates a selection of bits W down to X from bit vector U. $\{Y, Z\}$ indicates a concatenation of the bits of Y above Z. Thus, $U'16$ is a 16-bit field, and $\{U[7:0], U[15:8]\}$ indicates a byte-swap of U.

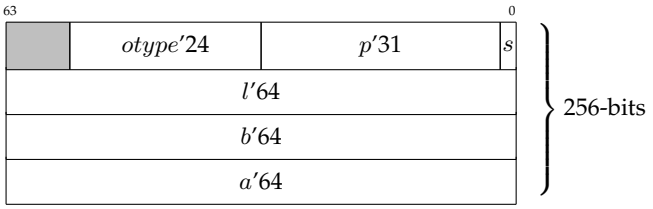
Furthermore, we use lower-case letters for full-length values (*e.g.*, a for address and t for top); we use upper-case letters for fields used for compression (*e.g.*, T for select bits of the top (t), and E for the exponent).

2.1 CHERI-256

The CHERI instruction-set architecture [1], [13] uses a large 256-bit capability format that encodes the base, length, and address as independent 64-bit fields. The simplicity of 64-bit integers is attractive, as complex encoding adds latencies to common operations. Intel MPX also uses a 256-bit format with base, top, and address as full 64-bit values [7], enabling simple low-latency bounds checking but wasting memory. We have built upon the uncompressed CHERI implementation shown in Figure 1, which we refer to as CHERI-256.

CHERI-256 supports *out-of-bounds pointers*; that is, the address may stray outside of bounds during address calculations with bounds enforced only on dereference. The term *out-of-bounds pointers* carries this meaning throughout this paper. CHERI-256 naturally supports such promiscuous arithmetic as the upper and lower bounds are independent of the address, each fully represented with 64-bit values that are unperturbed by wild modifications to the address field.

Previous fat-pointer systems have found that representing out-of-bounds pointers was necessary for compatibility with a broad code base [6], [15], and CHERI-256 relies on this feature to compile and execute a considerable amount of legacy C code [12], [13]. Despite these benefits, 256-bit pointers exact a heavy toll on cache footprint and processor data-path size. While we seek to encode fat-pointers more efficiently, we maintain both reasonable latencies for pointer arithmetic and out-of-bounds pointers.



otype: object type *p*: permissions *s*: sealed
l: object length *b*: base *a*: pointer address

Fig. 1: CHERI-256 capability format

2.2 M-Machine

The M-Machine [2] is a highly efficient capability-pointer design developed in the early 1990s. The M-Machine was not designed to support legacy software, but its capability format is elegantly simple and encodes base, top, and pointer address within 64 bits.

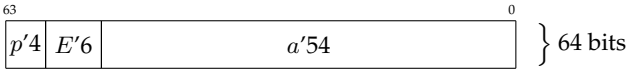


Fig. 2: M-Machine capability format

The M-Machine format, shown in Figure 2, encodes the base as the bits of a above E ($a[53:E]$) with the lower bits set to zero, the top as the base plus 2^E, and the current pointer address as simply a in its entirety. This capability-pointer compression introduces limitations on pointer arithmetic: address modifications must not change the decoded bounds without invalidating the capability pointer. For the M-Machine, pointer arithmetic may change only the bits below E, but any modification of the bits above E changes the decoded base, and must not produce a valid capability. Thus, all valid capability pointers maintain their original bounds, and out-of-bounds pointers cannot be represented.

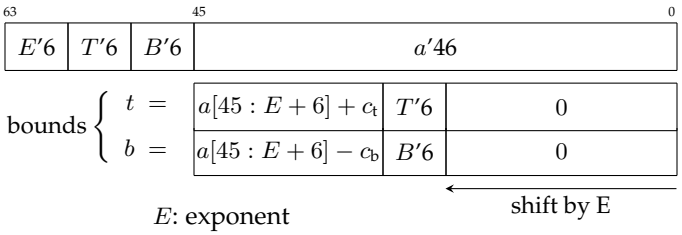
The M-Machine supports segments that are naturally 2^E aligned and 2^E sized. This power-of-two alignment restriction prevents precise enforcement of irregular object sizes. While the M-machine could mitigate coarse-grained memory safety issues by padding large objects with unallocated pages, this results in severe memory fragmentation, as demonstrated in Section 4.3. Any imprecise fat-pointer encoding would need allocator support to ensure memory safety; memory savings from pointer compression must be balanced against waste, due to memory fragmentation.

Finally, the M-Machine supports an unusual 54-bit address space due to dedicating upper bits to encode the bounds. This non-standard address size can cause compatibility issues, and somewhat limits future address-space expansion. The C language allows integers to be stored in pointers (e.g., *intptr_t*); the CHERI C compiler enables this behavior by placing the integer in the address field.

2.3 Low-fat pointers

Similar to the M-Machine, the Low-fat pointer scheme (*'Low-fat'*) compresses its capabilities into 64-bit pointers [3], with just 46 bits of usable address space (Figure 3). As in the M-Machine, memory is described in 2^E-sized blocks; however, regions may be described as a contiguous range of blocks. Thus, Low-fat supports a finer granularity for bounds than the M-Machine.

To encode a contiguous segment, Low-fat stores the 6-bit top (T) and base (B) blocks of the region, with the 6-bit block size, or exponent (E). T or B is simply inserted into the pointer address at E to produce the corresponding bound of the region, as illustrated in Figure 3. Low-fat can encode any span of blocks, up to a length of 2^6, regardless of alignment, by inferring any difference in the upper bits of the bounds. As a result, the only restriction on Low-fat regions is that the top and base must be aligned at 2^E.



E: exponent
T & B: top & base block number
a: address

$A_{\text{mid}} = a[E+5:E]$
$c_{\text{t}} = (A_{\text{mid}} \geqslant T)\,?\,1:0$
$c_{\text{b}} = (A_{\text{mid}} \leqslant B)\,?\,1:0$

Fig. 3: Low-fat capability format (our notation)

3 Shortcomings of the State-of-the-Art

The Low-fat work, an improvement over the similar Aries [14] scheme, presents an attractive middle ground between the restrictive M-Machine encoding and the verbose CHERI-256 approach. The decoding simplicity of Low-fat is promising; however, it still does not fit naturally into conventional pipelines, and is not compatible with common language semantics. CHERI Concentrate clears these hurdles, while also improving encoding efficiency.

3.1 Encoding Inefficiencies

Low-fat misses two opportunities for a more efficient encoding. First, if the exponent E is always chosen to be as small as possible, E directly implies the most significant bit of the size. This principle is used to save a bit in IEEE floating-point formats; with careful thought, we can save a bit in region encodings as well (see Section 4.2).

Second, Low-fat devotes equal encoding space for all values of E. That is, small regions and big regions have the same number of bits in T and B, despite the fact that small objects are far more common than large allocations.

3.2 Pipeline Problems

The Low-fat encoding requires all valid capability pointers to be in-bounds; therefore, pointer arithmetic must never produce an out-of-bounds pointer. This implies a bounds-check on all pointer arithmetic. A simple bounds-check requires decoding the bounds and comparing the bounds against the arithmetic result. Performing a pointer add, decoding the bounds, and the final comparison must all be completed after forwarded operands are available. Crucially, the comparison would need to be done after arithmetic is complete, extending the critical path of this logic.

The published implementation of Low-fat solves these issues by decoding capability pointers in the register file to eliminate additional delay for pointer arithmetic. The register file does not directly hold the decoded bounds, but holds the distance from the current address to the base and to the top of the region. These distances can be compared to the offset operand directly, in parallel with pointer addition, which results in no additional delay for the bounds check. This optimization has three costs:

- Delaying pointer loads due to decoding;
- Widening the register file to 164 bits;
- Updating the offsets on pointer arithmetic.

More than doubling the width of the register file for no architecturally visible benefit is undesirable; also, unpacking pointers on loads from memory is detrimental to performance. Load-to-use delay is a key performance parameter, and rarely has slack in a balanced design. Low-fat attempts to mitigate this issue by making the bounds available later in the pipeline than the address, although this introduces undue complexity; we demonstrate that there is a more efficient solution.

3.3 Out-of-Bounds Pointers being Unrepresentable

The authors of Low-fat note that their system can accommodate C-pointer calculations going out of bounds by padding allocations with unusable space – i.e., by simply widening the bounds beyond what was requested, and tagging the padded space as unusable with a separate fine-grained memory-type mechanism. Ideally, we would neither sacrifice memory nor require another complex mechanism to accommodate temporarily out-of-bounds pointers. While addresses should be allowed to wander temporarily beyond the bounds during pointer arithmetic, strict segment bounds should be enforced on dereference.

Both M-Machine and Low-fat invalidate pointers when arithmetic temporarily pushes them out of bounds, with the result that all valid pointers are in-bounds and no bounds check is required on dereference. While conceptually attractive, this optimization is neither realistic nor necessary. Dereference without a bounds check means that the instruction set (ISA) cannot support indexed addressing; while avoided in the bespoke Low-fat ISA, indexed addressing is required by every widely used ISA. Memory access already supports exceptions due to address translation, and any bounds check on the virtual address can be performed in parallel to translation, making memory access a particularly convenient time to perform a bounds check. In contrast, pointer-arithmetic operations are far more timing sensitive.

3.4 No Evaluation of Compiled Programs

Finally, the Low-fat pointer work was implemented within a proprietary instruction set without a TLB or even exception support, and thus was unable to validate support for compiled languages or an operating system. Indeed, no previous systems with capability-pointer compression have evaluated compiled programs. This leaves obvious questions, such as the frequency of various capability-pointer operations, as well as the appropriate granularity for bounds. While Low-fat is promising, its utility, as written, for general-purpose computing has yet to be demonstrated.

4 CC Principles I — Improving on Low-fat

CHERI Concentrate (CC) includes innovations in encoding efficiency, execution efficiency, and semantic flexibility. We describe an 18-bit encoding for bounds for direct comparison with Low-fat, and use this bounds field in our CHERI-64 encoding in Figure 9. However, the principles are independent of the field size; our 128-bit implementation (which supports a full 64-bit address space (Figure 13)) uses a 41-bit field for high-precision bounds. This section introduces innovations that directly improve the Low-fat model before introducing support for CHERI semantics in Section 5. A complete specification for decoding 64-bit CC (CHERI-64) is given in Figure 9.

4.1 Implied Most-Significant Bit of Top

If we consistently choose the smallest possible E to encode any set of bounds, the most significant bit of the length will be implied directly by E. Thus, we designed our instruction that encodes bounds (`CSetBounds`) to deterministically choose the smallest possible E when assigning an encoding from a full-precision base and length. All capabilities in the system are in this *normal form*.

For capabilities in the normal form, we can derive the top bit of T from the remainder of the encoding. We may conceptually imagine a 6-bit Length field, $L = (t-b)[E+5:E]$, where $T = B+L$. As the top bit of L is known to be 1, to calculate the top bit of T, we need only the top bit of B and the carry-out from the lower bits of $B + L$. This carry-out is 1 if $T[4:0] < B[4:0]$ – that is, when adding the lower bits of L to B has produced a value smaller than B. For the format in Figure 4, the formula to reconstitute the MSB of Top is

$$
\begin{aligned}
L_{\text{carry\_out}} &= \begin{cases} 1, & \text{if } T[4:0] < B[4:0] \\ 0, & \text{otherwise} \end{cases} \\
L[5] &= 1 (\text{implied}) \\
T[5] &= B[5] + L_{\text{carry\_out}} + L[5]
\end{aligned}
$$

Thus, as all capabilities in the system are in normal form, one bit can be saved in the encoding. The improved Low-fat format in Figure 4 uses this bit to indicate an *internal exponent*, as described in the Section 4.2.

4.2 Internal Exponent Encoding

As exponents of zero are most common, we encode a zero exponent with one bit (internal exponent, I_E), allowing 8-bit precision for small objects, improving on Low-fat for the common case. For larger objects, the lower bits of the bounds are used for a 6-bit exponent field.

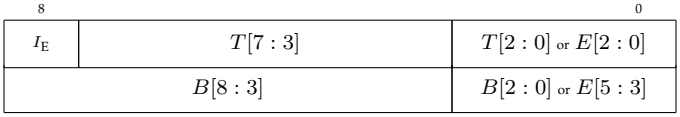


Fig. 4: Bounds with Embedded Exponent and Implied T_8

The most-significant bit of T is implied to be 1 only when E is nonzero. For all objects with sizes between 2^9 and 2^{64}, I_E is set, $T[8]$ is implied, and $T[2:0] = B[2:0] = 0$, leaving 6 bits of precision with a 5-bit T and 6-bit B field. In summary, CC can encode 8 bits of precision for small objects and 6 bits of precision for all others in the same 18 bits used by Low-fat, which offers a uniform 6 bits of precision.

4.3 Evaluation of Representability

In order to evaluate the usable precision of CC against the Low-fat encoding, we used the dtrace framework on Mac OS X 10.9 to collect traces from every allocator found in six real-world applications: Chrome 38.0.2125, Firefox 31, Apache 2.4, iTunes 12, MPlayer build #127, and mySQL 5. Allocators included many forms of *malloc()*, several application-specific allocators, driver internal allocators, and many other variants. We eliminated duplicate entries in the trace due to allocators passing the same requested allocation down through multiple lower-level allocators. Figure 5 shows the precision required for several sizes that would lose precision in Low-fat. While Figure 5 broadly justifies a 6-bit precision, nearly 2% of all allocations have a 7-bit length and require 7 bits of precision and are representable in CC but not Low-Fat. There are also notable collections of allocations that require up to 11 bits of precision, indicating the utility of the greater precision available in a 128-bit format.

Figure 6 gives representability results for specific applications. CC improves the precision of capabilities over Low-fat without increasing the number of bits required to encode bounds. As is clear from the encoding, CHERI CC is a strict improvement, i.e., under no circumstance does CHERI CC have worse precision than Low-fat.

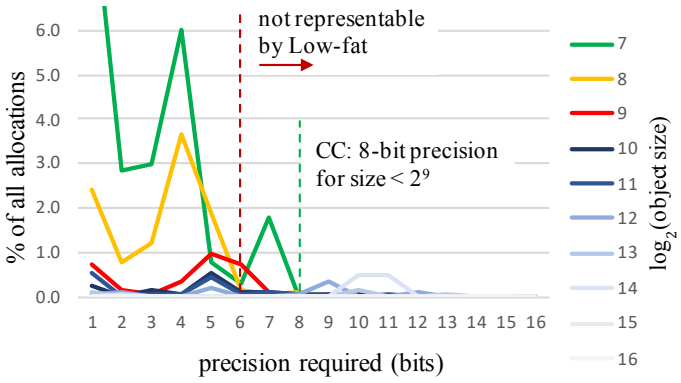


Fig. 5: Percentage of total allocations vs. precision required for a set of requested lengths for applications in Figure 6.

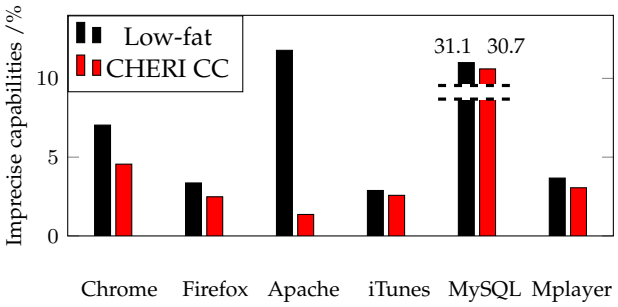


Fig. 6: The percentage of allocations that cannot be precisely represented in a capability. Lower is better.

4.4 Heap Allocators and Imprecision

The prevalence of small allocations is reflected in the design of the FreeBSD default memory allocator, `jemalloc()`, which assumes that applications will primarily allocate objects under 512 bytes [16]. When object bounds cannot be precisely represented by CC, the allocator may have to pad the allocation with unused memory to maintain memory safety. In practice, we have never observed `jemalloc()` requiring more than 6 bits to represent the memory reserved for an allocation, as the allocator itself requires alignment to ease memory management. Nevertheless, additional precision can be used to enforce precise object bounds, and enable precise enforcment of subobjects where possible.

5 CC Principles II — CHERI Semantics

Up to this point we have presented encoding improvements that directly translate to the Low-fat model. That is, we improved the encoding of a memory region in a 46-bit virtual address space where the address is between bounds. From this point we introduce some CHERI semantics necessary to support a legacy software base.

5.1 Full Address Space

CHERI semantics require a full 32-bit address space for 32-bit architectures, and 64-bit address space for 64-bit architectures. CHERI seeks to replace pointers in traditional computer systems with capability fat-pointers and to interact naturally with traditional operating systems and software. A non-standard address space, such as Low-fat's 46-bit proposal, would require a deeper rewrite of the modern software stack. CHERI Concentrate therefore chooses to support a 32-bit address space with our 18-bit bounds field in a 64-bit capability format, which we call CHERI-64. The smaller virtual address space reduces our E field by one bit, yielding the final 18-bit CHERI Concentrate format in Figure 7, which provides the same precision as the format in Figure 4, but also guarantees out-of-bounds representable space at least as large as the object itself. We support a full 64-bit address space with CHERI-128 described in Section 6.5.

5.2 Permissions Bits

Unlike Low-fat, CHERI supports permissions bits on capabilities for read, write, and execute, and to limit capability propagation, with a few permission bits reserved for software interpretation. CHERI Concentrate includes 12 permission bits in CHERI-64 and 15 bits in CHERI-128 to support the full CHERI permissions model.

5.3 Representable Buffer

CHERI-256 supports out-of-bounds pointers, encoding full, independent 64-bit words for the top, base, and address, to allow arbitrary pointer arithmetic without losing bounds [12]. This feature enables CHERI to substitute capabilities for pointers in a wide array of C programs without violating programmer expectations, providing memory safety without requiring unnecessary source modifications.

To confirm that out-of-bounds pointers are required to run a significant software stack, we implemented our capability format with the Low-fat strict bounds semantics. That is, we invalidated capabilities that went beyond object bounds so that they could no longer be dereferenced. Several binaries failed to run with these Low-fat semantics. Zlib provided the following critical example in `inftrees.c`:

```
static const unsigned short lbase[31] = ...;
...
base = lbase;
base -= 257;
...
val = base[work[sym]];
```

This function fails without temporarily out-of-bounds support because subtraction from `base` moves it well below the base of the object, though it actually should continue safely as `base` is later dereferenced legally using an offset larger than 257. Support for temporarily out-of-bounds pointers allows all binaries and libraries depending on zlib (including gzip, OpenSSH, libpng, etc.) to function as intended with compressed capabilities.

While out-of-bounds pointer support is desirable, our bounds are encoded with respect to the current pointer address, and this encoding cannot support unlimited manipulation without losing the ability to decode the original bounds. Nevertheless, we enable the vast majority of common behaviors for CC by extending the B field by 1 bit

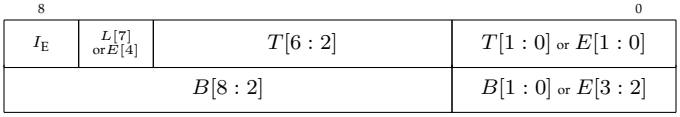


Fig. 7: 18-bit CHERI Concentrate Encoding with Representable Buffer Supporting a 32-bit Address Space

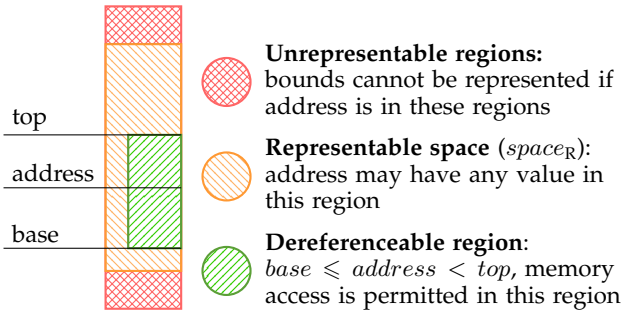


Fig. 8: Memory regions implied by a CC encoding

to provide a representable space that is at least twice the size of the object itself. This bit allows us to locate the base and top addresses with respect to the pointer address as the pointer moves beyond the object bounds. As a result, there are three address categories for any capability. Those between the bounds are in the *dereferenceable* region. These addresses are a subset of those within the larger *representable* space, $space_{\mathrm{R}}$. Addresses outside of $space_{\mathrm{R}}$ render the region *unrepresentable*, as depicted in Figures 8 and 10. With an extra bit of B to extend the representable space, we may now say that we infer the two most-significant bits of T:

$$L_{\text{carry\_out}} = \begin{cases} 1, & \text{if } T[6:2] < B[6:2] \\ 0, & \text{otherwise} \end{cases}$$

$$L_{\text{msb}} = \begin{cases} 1, & \text{if } I_{\mathrm{E}} = 1 \\ L_7, & \text{otherwise} \end{cases}$$

$$T[8:7] = B[8:7] + L_{\text{carry\_out}} + L_{\text{msb}}$$

6 CHERI Concentrate Region Arithmetic

We have carefully balanced the binary arithmetic needed for the CHERI Concentrate encoding to allow practical use in a traditional RISC pipeline. All operations that are traditionally single-cycle for integer pointers are also single-cycle in CHERI Concentrate despite enforcing sophisticated guarantees. These arithmetic operations are a crucial contribution of this work.

6.1 Encoding the bounds

We have added a `CSetBounds` instruction to the CHERI instruction set to allow selecting the appropriate precision for a capability.[1] `CSetBounds` takes the full pointer address a as the desired base, and takes a length operand from a general-purpose register, thus providing full visibility of the precise base and top to a single instruction – which can select the new precision without violating a tenet of MIPS (our base ISA) by requiring a third operand.

Deriving E

The value of E is a function of the requested length, l:

$$\text{index\_of\_msb}(x) = \text{size\_of}(x) - \text{count\_leading\_zeros}(x)$$
$$E = \text{index\_of\_msb}(l[31:8])$$

This operation chooses a value for E that ensures that the most significant bit of l will be implied correctly. If l is larger than 2^8, the most significant bit of l will always align with $T[7]$, and indeed $T[7]$ can be implied by E. If l is smaller than 2^8, E is 0, giving more bits to T and B and so enabling proportionally more out-of-bounds pointers than otherwise allowed for small objects.

We may respond to a request for unrepresentable precision by extending the bounds slightly to the next representable bound, or by throwing an exception. These two behaviors are implemented in the `CSetBounds` and `CSetBoundsExact` variants respectively.

1. Prior to this instruction, the bounds of a capability were set sequentially using `CIncBase` and `CSetLength`. `CIncBase` had to assign a compressed encoding to the base, possibly losing precision before the desired length was known, and `CSetLength` had no way to restore the lost precision if the final length would have allowed it.

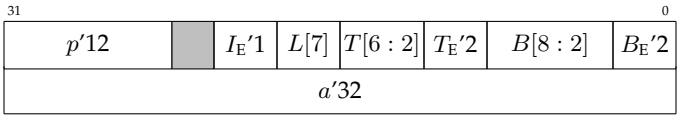
Extracting T *and* B

The `CSetBounds` instruction derives the values of B and T by simply extracting bits at E from b and t respectively (with appropriate rounding):

$E = 0$	$E > 0$; $T[1:0]$ and $B[1:0]$ implied 0s
$T = t[6:0]$	$T[6:2] = t[E+6:E+2] + \text{round}$ $\text{round} = \text{one\_if\_nonzero}(t[E+1:0])$
$B = b[8:0]$	$B[8:2] = b[E+8:E+2]$

Rounding Up $length$

The `CSetBounds` instruction may round up the top or round down the base to the nearest representable alignment boundary, effectively increasing the length and potentially increasing the MSB of length by one, thus requiring that E increase to ensure that the MSB of the new L can be correctly implied. Rather than detect whether overflow will certainly occur (which did not pass timing in our 100MHz CHERI-128



p: permissions I_{E}: internal exponent a: address

If $I = 0$:

$$\begin{aligned} E &= 0 \\ T[1:0] &= T_{\mathrm{E}} \\ B[1:0] &= B_{\mathrm{E}} \\ L_{\text{carry\_out}} &= \begin{cases} 1, & \text{if } T[6:0] < B[6:0] \\ 0, & \text{otherwise} \end{cases} \\ L_{\text{msb}} &= L_7 \\ T[8:7] &= B[8:7] + L_{\text{carry\_out}} + L_{\text{msb}} \end{aligned}$$

If $I = 1$:

$$\begin{aligned} E &= \{L_7, T_{\mathrm{E}}, B_{\mathrm{E}}\} \\ T[1:0] &= 0 \\ B[1:0] &= 0 \\ L_{\text{carry\_out}} &= \begin{cases} 1, & \text{if } T[6:2] < B[6:2] \\ 0, & \text{otherwise} \end{cases} \\ L_{\text{msb}} &= 1 \\ T[8:7] &= B[8:7] + L_{\text{carry\_out}} + L_{\text{msb}} \end{aligned}$$

t =	$a[31:E+9] + c_{\mathrm{t}}$	$T[8:0]$	$0'E$
b =	$a[31:E+9] + c_{\mathrm{b}}$	$B[8:0]$	$0'E$

shift by E

To calculate c_{t} and c_{b}:

$$A_{\text{mid}} = a[E+8:E]$$
$$R = \{B[8:6] - 1, zeros'6\}$$

$A_{\text{mid}} < R$	$T < R$	c_{t}
false	false	0
false	true	+1
true	false	−1
true	true	0

$A_{\text{mid}} < R$	$B < R$	c_{b}
false	false	0
false	true	+1
true	false	−1
true	true	0

Fig. 9: CC capability format

FPGA prototype), we choose to detect whether $L[7:3]$ is all 1s – i.e., the largest length that would use this exponent – and force T to round up and increase E by one. This simplifies the implementation at the expense of precision for $1/16$th of the requestable length values.

6.2 Decoding the bounds

Unlike Low-fat, CHERI Concentrate can decode the full t and b bounds from the B and T fields even when the pointer address a is not between the bounds. We now detail how each bit of the bounds is produced:

Lower bits: The bits below E in t and b are zero, that is, both bounds are aligned at E.

Middle bits: The middle bits of the bounds, $t[E+8:E]$ and $b[E+8:E]$, are simply T and B respectively, with the top two bits of T reconstituted as in Section 5.3. In addition, if I_{E} is set, indicating that E is stored in the lower bits of T and B, the lower two bits of T and B are also zero.

Upper bits: The bits above $E+8$, for example $t[31:E+9]$, are either identical to $a[31:E+9]$, or need a correction of ± 1, depending on whether a is in the same alignment boundary as t, as described below and in Figure 9.

Deriving the representable limit, R

CC allows pointer addresses within a power-of-two-sized space, $space_{\mathrm{R}}$, without losing the ability to decode the original bounds. The size of $space_{\mathrm{R}}$ is $s = 2^{E+9}$, fully utilizing the encoding space of B. Figure 10 shows an example of object bounds within the larger $space_{\mathrm{R}}$. Due to the extra bit in B, $space_{\mathrm{R}}$ is twice the maximum object size (2^{E+8}), ensuring that the out-of-bounds representable buffers are, in total, at least as large the object itself.

As portrayed in Figure 10, $space_{\mathrm{R}}$ is not usually naturally aligned, but straddles an alignment boundary. Nevertheless, as $space_{\mathrm{R}}$ is power-of-two-sized, a bit slice from its base address $r_b[E+9:E]$ will yield the same value as a bit slice from the first address above the top, $r_t[E+9:E]$. We call this value the *representable limit*, R. Locating b, t, and a either above or below the alignment boundary in $space_{\mathrm{R}}$ requires comparison with this value R. We may choose R to be any out-of-bounds value in $space_{\mathrm{R}}$, but to reduce comparison logic we have chosen:

$$R = \{B[8:6] - 1, zeros'6\}$$

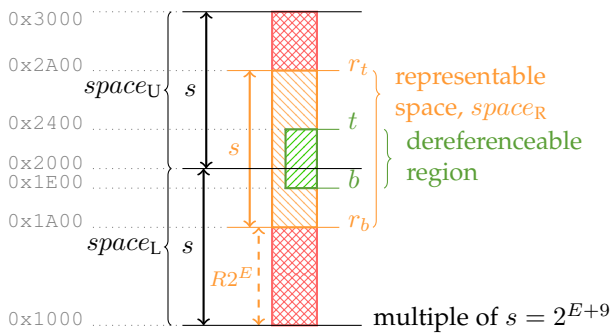


Fig. 10: CHERI Concentrate bounds in an address space. Addresses increase upwards. To the left are example values for a `0x600`-byte object based at `0x1E00`.

This choice ensures that R is at least $1/8$ and less than $1/4$ of the representable space below b, leaving at least as much representable buffer above t as below b.

For every valid capability, the address a as well as the bounds b and t lie within $space_{\mathrm{R}}$. However the upper bits of any of these addresses may differ by at most 1 by virtue of lying in the upper or lower segments of $space_{\mathrm{R}}$. For example, if a is in the upper segment of $space_{\mathrm{R}}$, the upper bits of a bound will be one less than the upper bits of a if the bound lies in the lower segment. We can determine whether a falls into upper or lower segment of $space_{\mathrm{R}}$ by inspecting:

$$A_{\mathrm{mid}} = a[E+8:E]$$

If A_{mid} is less than R, then a must lie in the upper segment of $space_{\mathrm{R}}$, and otherwise in the lower segment. The same comparison for T and B locates each bound uniquely in the upper or the lower segment. These locations directly imply the correction bits c_{t} and c_{b} (computed as shown in Figure 9) that are needed to compute the upper bits of t and b from the upper bits of a. As we have chosen to align R such that $R[5:0]$ are zero, only three-bit arithmetic is required for this comparison, specifically:

$$a\_in\_upper\_segment = A_{\mathrm{mid}}[8:6] < R[8:6]$$

While Low-fat requires a 6-bit comparison to establish the relationship between a, t, and b, growing with the precision of the bounds fields, CC requires a fixed 3-bit comparison regardless of field size, particularly benefiting CHERI-128, which uses 21-bit T and B fields. CC enables capabilities to be stored in the register file in compressed format, often requiring decoding before use. As a result, this comparison lies on several critical paths in our processor prototype.

The bounds t and b are computed relative to A_{upper}:

$$t = \{(A_{\mathrm{upper}} + c_{\mathrm{t}}), T, zeros'E\}$$
$$b = \{(A_{\mathrm{upper}} + c_{\mathrm{b}}), B, zeros'E\}$$
$$\text{where } A_{\mathrm{upper}} = a[31:E+9]$$

The bounds check during memory access is then:

$$b \leqslant computed\_address < t$$

In summary, CC generalizes Low-fat arithmetic to allow full use of the power-of-two-sized encoding space for representing addresses outside of the bounds, while improving speed of decoding.

Encoding full address space

The largest encodable 32-bit value of t is `0xFF800000`, making a portion of the address space inaccessible to the largest capability. We can resolve this by allowing t to be a 33-bit value, but this bit-size mismatch introduces some additional complication when decoding t. The following condition is required to correct t for capabilities whose representable region wraps the edge of the address space:

$$\textbf{if } ((E < 24)\ \&((t[32:31] - b[31]) > 1)) \textbf{ then } t[32] = !t[32]$$

That is, if the length of the capability is larger than E allows, invert the most significant bit of t.

6.3 Fast representable limit checking

Pointer arithmetic is typically performed using addition, and does not raise an exception. If we wish to preserve these semantics for capabilities, capability pointer addition must fit comfortably within the delay of simple arithmetic in the pipeline, and should not introduce the possibility of an exception. For CC, as with Low-fat, typical pointer addition requires adding only an offset to the pointer address, leaving the rest of the capability fields unchanged. However, it is possible that the address could pass either the upper or the lower limits of the representable space, beyond which the original bounds can no longer be reconstituted. In this case, CC clears the tag of the resulting capability to maintain memory safety, preventing an illegal reference to memory from being forged. This check against the representable limit, R, has been designed to be much faster than a precise bounds check, thereby eliminating the costly measures the Low-fat design required to achieve reasonable performance.

To ensure that the critical path is not unduly lengthened, CC verifies that an increment i will not compromise the encoding by inspecting only i and the original address field. We first ascertain if i is *inRange*, and then if it is *inLimit*. The *inRange* test determines whether the magnitude of i is greater than that of the size of the representable space, s, which would certainly take the address out of representable limits:

$$inRange = -s < i < s$$

The *inLimit* test assumes the success of the *inRange* test, and determines whether the update to A_{mid} could take it beyond the representable limit, outside the representable space:

$$inLimit = \begin{cases} I_{\text{mid}} < (R - A_{\text{mid}} - 1), & \text{if } i \geqslant 0 \\ I_{\text{mid}} \geqslant (R - A_{\text{mid}}) \text{ and } R \neq A_{\text{mid}}, & \text{if } i < 0 \end{cases}$$

The *inRange* test reduces to a test that all the bits of I_{top} ($i[63 : E+9]$) are the same. The *inLimit* test needs only 9-bit fields ($I_{\text{mid}} = i[E+8, E]$) and the sign of i.

The I_{mid} and A_{mid} used in the *inLimit* test do not include the lower bits of i and a, potentially ignoring a carry in from the lower bits, presenting an *imprecision hazard*. We solve this by conservatively subtracting one from the representable limit when we are incrementing upwards, and by not allowing any subtraction when A_{mid} is equal to R.

One final test is required to ensure that if $E \geqslant 23$, any increment is representable. (If $E = 23$, the representable space, s, encompases the entire address space.) This handles a number of corner cases related to T, B, and A_{mid} describing bits beyond the top of a virtual address. Our final fast *representability* check composes these three tests:

$$representable = (inRange \text{ and } inLimit) \text{ or } (E \geqslant 23)$$

To summarize, the representability check depends only on four 9-bit fields, T, B, A_{mid}, and I_{mid}, and the sign of i. Only I_{mid} must be extracted during execute, as A_{mid} is cached in our register file. This operation is simpler than reconstructing even one full bound, as demonstrated in Section 8. This fast representability check allows us to perform pointer arithmetic on compressed capabilities directly, avoiding decompressing capabilities in the register file that introduces both a dramatically enlarged register file and substantial load-to-use delay.

6.4 Examplary Encodings

We walk through a pair of exemplary capability encodings to illustrate the above encoding details. In Figures 11 and 12, the dark fields were requested by `CSetBounds`, the orange and green fields are stored in the capability encoding, and the white fields are implied by the encoded fields. Each example shows only the bottom 12 bits of each field.

Unaligned 128-byte example

A programmer has instantiated a 129-character string on his stack, and wants to trim the first character and pass the subset to a function. The capability instructions to create the subset are as follows:

```
CIncOffset $trimmed, $str, 1
CSetBounds $trimmed, $trimmed, 128
```

As this string began on the stack that is at least word-aligned, the *trimmed* capability is now unaligned. The requested capability is perfectly representable using CC, as are all objects less than 255 bytes; the resulting encoding is shown in Figure 11. We note that this capability would not be representable in Low-Fat, which can represent with byte-precision only up to 63-byte objects.

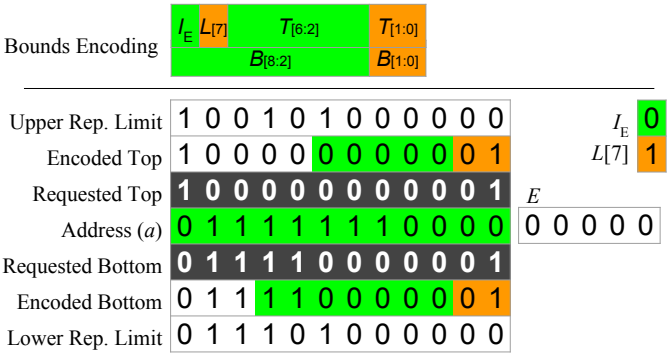


Fig. 11: Example Capability with Exponent=0

Spanning alignment boundary: We observe that the upper bits of the Top are entirely different from the upper bits of Bottom in Figure 11, though they numerically differ by only 1. During decode, we can ascertain that the upper bits of Top must be one larger than the upper bits of Bottom using the following steps:

$$T[8:7] = B[8:7] + L[7] + L_{\text{carry\_out}} \quad (1)$$
$$= 11 + 01 + 00 = 00 \quad (2)$$

This produces a $T[8:0]$ of 000000001. We may now infer any difference in the bits above $T[8]$ in the pointer using the representable limit, R:

$$A_{\text{mid}} = a[8:0] = 111110000 \quad (3)$$
$$R_{\text{u}} = R[8:6] = B[8:6] - 001 = 101 \quad (4)$$
$$c_{\text{t}} = \begin{cases} 0, & \text{if } (R_{\text{u}} < T[8:6]) = (R_{\text{u}} < A_{\text{mid}}[8:6]) \\ 1, & \text{otherwise} \end{cases} \quad (5)$$
$$T[11:9] = A_{\text{upper}}[2:0] + c_{\text{t}} = 011 + 001 = 100 \quad (6)$$

504-byte example

Let us now consider a larger object that cannot be represented precisely with CC, and that has an out-of-bounds pointer:

```
char str[504];
skip16(str − 16);
```

The encoding of the new object is shown in Figure 12 and the assembly that will generate the capability passed to skip16 is:

```
CSetBounds $str, $sp, 504
CIncOffset $str, $str, −16
```

Fig. 12: Example Capability with Encoded Exponent

Losing precision: The size of this object is 504 bytes, which is larger than the maximum size of 255 that can be precisely represented by CC. We select E by inspecting the length that has been requested, which is 111111000 in binary.

$$E_t = \text{index\_of\_msb}(l[11:8]) = 1 \tag{7}$$

$$E = \begin{cases} E_t + 1, & \text{if } L[7:4] = 1111 \\ E_t, & \text{otherwise} \end{cases} \tag{8}$$

$$E = 2 \tag{9}$$

In this case, due to the length being the maximum possible length we could represent with this exponent, we increase E by one to account for the possiblity that the length may round up. Since E is non-zero, we must encode the exponent in the capability ($I_E = 1$), and $T[1:0]$ and $B[1:0]$ are no longer available for precision. As a result, the bottom four bits are rounded to the appropriate alignment boundary (away from the object) in the Encoded Top and Bottom in Figure 12, and the encoded length is 512 bytes.

Representability Check: After the bounds of the capability are set, we move the pointer 16 bytes below the base. We assert that the result of the add of -16 (111111110000) will be representable with the following steps:

$$inRange = -s < i < s = -2048 < -16 < 2048 \tag{10}$$

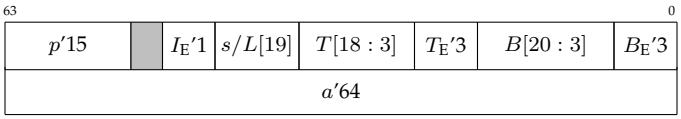
$$R - A_{\text{mid}} = 0010000 - 0100000 = 1010000 \tag{11}$$

$$inLimit = (I_{\text{mid}} \geqslant (R - A_{\text{mid}})) \,\&\, (R \neq A_{\text{mid}}), \text{as } i < 0 \tag{12}$$

$$= (1111111 \geqslant 1010000) \,\&\, (0010000 \neq 0100000) \tag{13}$$

$$rep. = inRange \,\&\, inLimit = \text{True} \tag{14}$$

As the subtraction of 16 will produce a result within representable bounds, we may simply perform the subtraction on the Address field of the capability, producing the result in Figure 12.



p: permissions s: sealed a: pointer address

Fig. 13: CHERI-128 capability format.

6.5 CHERI-128

While we have implemented the above format with 18 bits for bounds for a 32-bit address space in CHERI-64, our CHERI Concentrate format for a 64-bit address space uses 41 bits for bounds. This CHERI-128 format uses 21 bits for B and and 19 bits for T, and is shown in Figure 13. Our CHERI-64 and CHERI-128 encodings reserve a few bits for future use (2 bits and 7 bits respectively), which could be applied to greater precision if needed.

7 Instruction Frequency Study

CHERI Concentrate pipeline optimizations have a firm grounding in analysis of compiled capability programs. Table 1 contains the first published study of the frequency of capability instructions in compiled programs. These programs include the Duktape Javascript interpreter running the Splay benchmark from the Octane suite, a SQLite benchmark developed for the LevelDB project, the P7Zip benchmark from the LLVM test suite, and a boot of FreeBSD with all user-space processes compiled in a pure-capability mode. In each case we traced around 1 billion user-space instructions from the FPGA implementation, about 10 seconds of execution time on our 100MHz processor, sampled throughout the benchmark.

This study of capability instructions is unique compared to studies using conventional instruction sets, as a capability instruction set distinguishes between pointer and integer operations. A capability instruction set distinguishes between memory operations accessing non-pointer data and those that support pointers; it also distinguishes between pointer modification and integer arithmetic instructions. According to Table 1, pointer-sized loads constitute up to 12% of common programs. Thus, pointer loads should minimize additional delay caused by an *unpack* operation to decode capabilities into the register file or else risk

TABLE 1: Dynamic capability instruction mix

Percent of total instruction mix for different benchmarks				
Category	**DukJS**	**SQLite**	**P7Zip**	**Boot**
load/store data	13.69%	21.29%	16.93%	16.26%
load capability	11.69%	7.99%	1.52%	4.69%
store capability	6.91%	5.24%	0.62%	3.17%
cap pointer arithmetic	15.15%	13.16%	7.19%	2.82%
stack pointer	5.06%	3.35%	0.99%	0.93%
other	10.09%	9.81%	6.20%	1.89%
jump to capability	2.73%	1.68%	0.50%	0.50%
get special capability	0.08%	0.03%	0.00%	0.02%
compare capabilities	1.38%	1.90%	0.19%	0.15%
set capability bounds	0.36%	0.69%	0.00%	0.07%
read a capability field	1.17%	0.68%	0.00%	0.05%

greatly impacting performance. Table 1 further indicates that pointer arithmetic commonly constitutes over 10% of executed instructions. Therefore *pointer add* must remain simple, fast, and energy-efficient, in the face of new requirements imposed by capabilities. Table 1 shows loads and stores of data and capabilities constituting as much as 35% of common programs. Therefore the *bounds check* on the offset addressing operation must not impede the critical path. or else would risk greatly impacting program performance.

As described in Section 8, CHERI Concentrate respects all three of these requirements by nearly eliminating the *unpack* operation, by introducing a fast *representability check* for pointer arithmetic using only the compressed format, and by acknowledging that the more complex *bounds check* operation does not lie on the critical path.

8 CHERI Concentrate Pipeline

Capability compression adds three operations to a pipeline:

1) **Unpack**: Any logic that transforms a capability from memory representation to register format.
2) **Pointer Add**: Any logic required to add an offset to a capability, producing a new capability, handling any checks for an unrepresentable result.
3) **Bounds Check**: Any logic to verify that an offset lands within the bounds of a capability for memory access.

Figure 14 shows the placement of these operations in a typical MIPS pipeline using the Low-Fat and CHERI Concentrate micro-architectural approaches. In the CC pipeline, Unpack and Pointer Add lie on a performance-critical path, and Bounds Check does not. For Low-Fat, all three operations are moved to a new capability pipeline stage after memory access where bounds are unpacked, updated, and checked. Placing all bounds operations in a single stage after cache access enables full pipelining even as capabilities are loaded, as long as bounds are not required earlier in the pipeline. This is true as long as loads and stores can be issued with the pointer address speculatively, verifying bounds only in the cycle when a loaded value is available.

Unpack

Low-Fat has a complex Unpack operation that decodes the distance to the bounds into the register file. Complete Low-Fat bounds decoding required over 4ns on their Xilinx Virtex 6 [3] and 4.47ns in our experiment in Figure 15, which is too complex to be performed after memory load and before register writeback without requiring an additional cycle. Low-Fat added an Unpack stage to the pipeline and also performed Pointer Add and Bounds Check there to ensure that these operations can see capabilities forwarded from a load. It is preferable to eliminate this stage and maintain a single set of forwarding paths to Execute.

CHERI Concentrate does not entirely eliminate the Unpack procedure but reduces its cost dramatically so that it is not in the critical path. The CC unpack operation requires 1.70ns to decode the bits of the pointer address at Exp (corresponding to T and B) and the top two bits of T. This delay is comparable to the data byte select, which is not required when loading capabilities; thus, CC avoids extra delay between the cache and the register file.

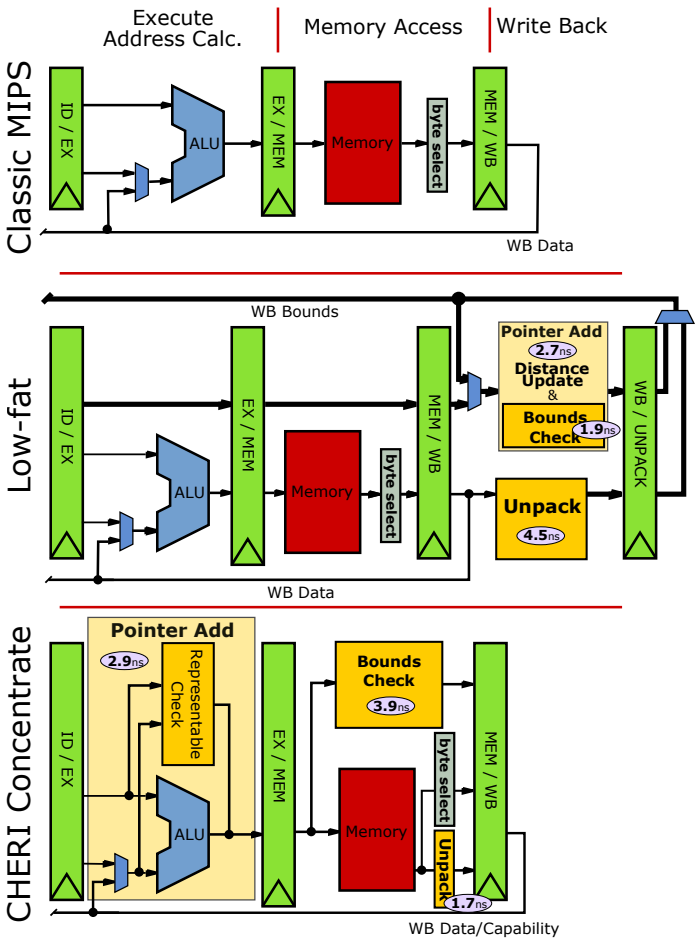


Fig. 14: Crucial Capability Functions in the Pipeline

Pointer Add

Low-Fat Pointer Add consists of 3 operations: address addition, Bounds Check, and a distance update. The address addition is done in the traditional ALU, but the Bounds Check and update of the distances to the bounds occur in the new pipeline stage for bounds. The Bounds Check operation is highly optimized by fully decoding the distance to the bounds in the register file, requiring only 1.88ns in our experiment in Figure 15, and resulting in a full Pointer Add of only 2.74ns – only 70% longer than the simple 64-bit add required by CHERI-256.

CC does not have the advantage of fully decoded bounds, but uses the *representability check* described in Section 6.3 to assert that the result will be representable without checking the precise bounds. This check achieves a delay of only 2.89ns to modify the pointer address and to perform the representability check – only slightly longer than Low-fat's Pointer Add. CC's representability check fits easily within the execute stage of our pipeline.

Bounds Check

A standalone *Bounds Check* operation is very fast for Low-Fat, only 1.88ns, but we consider this a wasted optimization as the stand-alone bounds check would not lie on the critical path for memory access. CC required 3.85ns for a precise *Bounds Check*, which fits comfortably into the exception path of our pipeline (which is parallel to cache lookup).

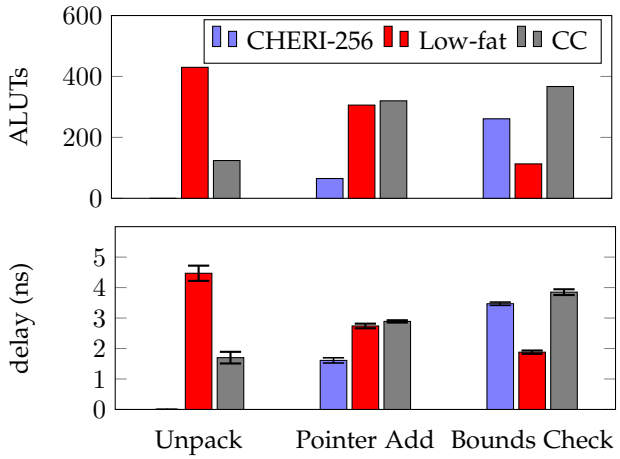


Fig. 15: Complexity and speed of key operations. Test bench synthesized in Altera's Quartus Prime 15.1 for a Stratix V 5SGXEA7N2F45C2 FPGA. Each synthesis performed with three random seeds. Combinational logic usage (ALUTs) was constant, but layout variation perturbed timing. The Low-fat algorithms were reproduced from their descriptions. The CC algorithm used here has a 48-bit virtual address and an 18-bit bounds field for direct comparison with Low-fat.

By pushing the full *Bounds Check* delay from the pointer load-to-use path to the memory access path, CHERI Concentrate avoids pointer load-to-use delays, an inflated register file, and unusual pipeline forwarding. While we performed our integration with a canonical MIPS pipeline, these solutions enable a reasonable capability implementation in any processor without violating the general conventions of high-performance pipelines.

9 Execution Performance

To evaluate the CC, we modified our open-source CHERI processor from http://www.cheri-cpu.org/, extended the LLVM compiler [17], created a custom embedded OS based on CHERI protection, and extended the FreeBSD OS [18].

9.1 Microbenchmarks

We used a modified LLVM to compile a number of small benchmarks to use capabilities for all data pointers (including heap and stack allocations) to enforce spatial memory safety, and to use capabilities for return addresses and function pointers to enforce Control-Flow Integrity (CFI) [19]. We included the MiBench [20] suite, which is representative of typical embedded data-centric C code, and the Olden [21] suite, which is representative of pointer-based data structure algorithms. These were executed under a custom embedded operating system running on a 100MHz Stratix IV FPGA prototype that used a 32KiB, 4-way set-associative, write-through L1 data cache and a 256KiB, 4-way write-back L2 cache. Across all benchmarks, we compare CHERI-256, CHERI-128 and CHERI-64. This study allows us to compare CHERI-256 with CHERI-128, which provides a direct improvement by halving pointer size while supporting the full 64-bit virtual address space with negligible alignment restrictions. On the other hand, we can compare CHERI-128 with CHERI-64, which restricts the virtual address space to 32 bits (based on the MIPS-n32 ABI), using the upper 32 bits to encode capability fields. The three architectures share code generation and differ only in capability size.

Figure 16 measures the improvement in execution time and L2 cache misses of CHERI-128 and CHERI-64 against CHERI-256, with CHERI-256 normalized at 100%. The two metrics represent overall performance and additional DRAM traffic respectively. We have ordered the graphs by pointer memory footprint as measured from core dumps. The box plots aggregate all benchmarks with a pointer density of less than 0.2% of allocated memory. These include bitcount, qsort, stringsearch, rijndael, CRC, SHA, dijkstra and adpcm, all from the MiBench suite. Apart from Patricia, MiBench has low pointer density and has little memory impact of using capabilities of any size. The Olden benchmarks with pointer-based data structures show up to a 20% reduction in run time and a 50% reduction in DRAM traffic when moving from CHERI-256 to CHERI-128. CHERI-64 further improves over CHERI-128, with a performance improvement approaching 10%, but achieving an equally dramatic reduction in DRAM traffic for these pointer-heavy use cases.

The low-pointer density benchmarks show almost no performance improvement with smaller pointers, indicating that many applications will see very little cost from adopting extensive capability protections regardless of capability size. Nevertheless, these low-pointer-density applications occasionally saw a notable decrease in L2 misses.

9.2 Larger applications and benchmark suites

We have designed CHERI Concentrate to fit in a standard RISC architecture and to support compiled C programs. As a result, we are able to compile many standard applications and execute them under a full operating system, CheriBSD (i.e., FreeBSD with capability extensions).

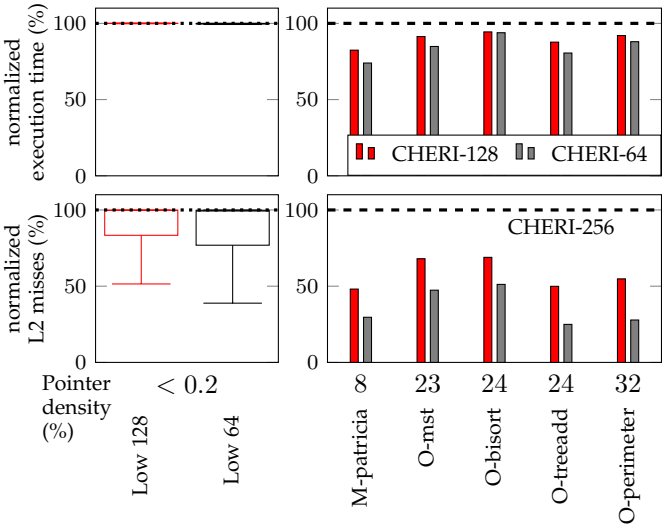


Fig. 16: Percentage of run time and total L2 misses for CHERI-128 and CHERI-64 versus CHERI-256 (dashed lines, normalized to 100%). MiBench benchmarks with low pointer density (Low 128/64) are collated on the left. High pointer-density benchmarks are on the right.

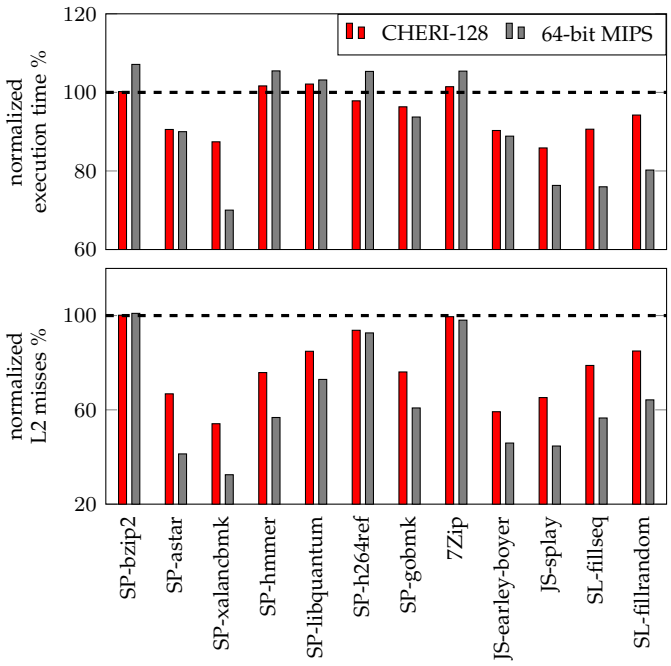


Fig. 17: Percentage of run time and total L2 misses for CHERI-128 and 64-bit MIPS versus CHERI-256 (dashed line, normalized to 100%): SPECint 2006 (SP-), Javascript (JS-), and Sqlite3 (SL-). Origins are non-zero to improve visibility.

This section uses the same hardware platform and compiler configuration as in Section 9.1. All benchmarks (P7Zip 16.02, Octane with Duktape 1.4.0, Sqlite3 3.21.0, and SPECint 2006) run under CheriBSD. SPECint 2006 benchmarks are run with test datasets due to time and memory constraints on our FPGA platform. Unfortunately, due to the lack of LLVM MIPS-n32 support under FreeBSD, the CHERI-64 results are replaced with classic 64-bit MIPS benchmarks as a reference. Note that MIPS code not only has smaller pointer sizes, but also does not have any overhead from CHERI protection at all. MIPS code generation also differs significantly from CHERI code generation when using integer registers for pointer access, which would cause CHERI to be slightly faster in some cases. This paper demonstrates the benefits of capability size reduction and does not speculate on optimal capability code generation, although the CHERI LLVM extension continues to improve.

Figure 17 shows the results obtained from these workloads. P7Zip is an ALU- and data-heavy benchmark with very few pointers, resulting in the performance of CHERI-256, CHERI-128 and MIPS being very close. The Octane JavaScript benchmarks, Splay and Earlay-Boyer, were selected for pointer density, with around 25% of all data memory holding pointers. These JavaScript benchmarks running under Duktape show a dramatic improvement with capability compression, cutting run time by 10% and reducing L2 cache misses by 40%. Though the common database application Sqlite3 has lower pointer density and relies heavily on file I/O, there is still a sizable reduction of run time and DRAM traffic. On average, approximately 10% of the total run time and 30% of the DRAM traffic can be eliminated by deploying CHERI-128 for these benchmarks.

10 Proof of Correctness

The CHERI Concentrate capability compression and decompression algorithms are complex. Consequently, we have undertaken a machine-checked proof of correctness in HOL4 for key properties, which identified bugs and confirmed the necessity of all corner-case handling required in the hardware implementation. The first four proofs relate to compression and decompression, and the last two proofs verify the fast representable bounds check. These proofs produced counterexamples for our initial E selection algorithm, and for our initial representability-check algorithm; the corrections are reflected in the algorithms presented in Section 6. These proofs are available online: ⟨www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-concentrate/⟩.

- For any address (a) in the representable region defined by a requested $base$ and top from which we derive an encoded b and t:
  1) $b \leqslant base$
  2) $base - b < 2^{E+2}$, that is, the error on the encoded base is less than the value of least significant bit of B when there is an internal exponent
  3) $t \geqslant top$
  4) $t - top \leqslant 2^{E+2}$, that is, the error on the encoded top is less than the value of the least significant bit of T when there is an internal exponent
- For any address a, increment i, valid exponent $E < 26$, and representable limit R (see Section 6.2):
  1) The fast representability check, *IsRep*, will succeed only if $p = a + i$ is within one representable space, $s = 2^{E+9}$, of the representable base r_b:

$$IsRep \implies p - r_b < s$$

  2) The fast representability check will succeed if p is reasonably within s of r_b:

$$2^E \leq p - r_b < s - 2^E \implies IsRep$$

11 Related Work

CHERI Concentrate safely encodes fine-grained memory protection properties similar to the M-machine and Low-Fat pointers, representing a family of capability fat-pointer machines. Computer architects have also explored several other useful approaches to encode memory protection metadata in computer systems.

11.1 Table-based encoding

Table-based designs encode protection information in an external table, keyed either by the data memory address, choice of segment registers, or by an explicit index in the memory reference. Whereas an arbitrarily large amount of protection metadata may be encoded in a small index, the table approach optimizes for a small fixed set of objects, and does not fit today's large and layered software landscape.

Early capability systems described a *C-list* of capabilities for a process [22]; systems including the CAP computer [8] and the i960 [10] implemented pointers as indices into this table. Some foundational capability systems, in addition to supporting tables of memory descriptor capabilities,

supported capabilities as abstract identifiers that required interpretation from a trusted object manager [23], [24].

Page-based memory protection is a table-based design that is ubiquitous in commercial hardware and usually includes protection metadata. Page-based protection has been extended in various ways to support in-address-space security domains including *domain-page protection* [25] and *page group identifiers* in the HP PA-RISC [26]. In page-based systems, protection metadata is associated with the virtual address and not with the pointer; costs scale with the size of the address space and not with the number of pointers. However, as a consequence any access to memory in a process is treated equally, and it is not possible to detect a corrupted pointer that illegally accesses valid memory in the process.

Mondriaan [4] also uses table indirection to encode protection metadata in a Protection Look-aside Buffer (PLB) for fine-grained address validation. While Mondriaan avoids page granularities, performance still benefits greatly from reducing the number of objects, making this approach undesirable for fine-grained protection.

Segmentation, pioneered in the Multics system [27] and once common on IA-32 architectures (now deprecated in Intel 64 [28]), encodes protection metadata in segment descriptor tables indexed by segment selector registers. Despite earlier wide-spread deployment, this memory protection primitive was not picked up in scalable language models.

11.2 Tagged memory

Complex tags on memory locations may also encode protection metadata for pointers in memory [29], [30], [31].

Silicon Secured Memory from Oracle [32] tags all data with *version* numbers, which must match in-pointer metadata to avoid temporal confusion.

Memory Protection Extensions (MPX) [7] from Intel maps a shadow space for protection metadata for pointers in memory, and is similar to *HardBound* [6] (an academic proposal). Despite hardware support, no form of compression is used, and the metadata space is four times the size of the protected address space.

As memory locality suffers from shadow-space or table lookups, CHERI minimizes the tag to a single bit, which can be used to protect the integrity of the remainder of the metadata that may be stored in-line.

11.3 Capability pointer compression

CHERI Concentrate is based on CHERI-256 and inspired by the M-Machine and Low-fat Pointers (which are thoroughly discussed in Section 2).

11.4 Software fat-pointer techniques

Software-only techniques for fine-grained protection have achieved surprising performance, but have had large memory overheads due to lack of compression. *Baggy bounds* [33], *SoftBound* [34] and *PAriCheck* [35] dynamically check bounds of fat pointers in software, and trade memory fragmentation for improved performance. *Cyclone* [36] explicitly breaks compatibility with C to define a safer C dialect that provides fine-grained memory safety. Cyclone's abstraction is close to the CHERI model, but adds many static annotations. Although Cyclone was not widely adopted, it influenced pointer annotation in current C compilers. CHERI Concentrate can accelerate such systems to allow precise checks, with negligible performance overhead.

12 Conclusions

CHERI Concentrate (CC) resolves major roadblocks to the adoption of capability fat-pointers for fine-grained, deterministic memory protection. CC inherits the mature CHERI capability semantics, enabling support for a wide software base. CC also inherits the efficient Low-Fat compression techniques, improving compression efficiency by eliminating one bit of the bounds without losing precision and by encoding the exponent within the bounds field, achieving the highest published capability fat-pointer encoding efficiency. In addition, we developed arithmetic operations that operate directly on the compressed encoding. These allow the register file to hold compressed capabilities, eliminating complexity on the load path, and also allow more flexible out-of-bounds modifications required by CHERI semantics.

We validated our design by building an FPGA prototype that achieves the same frequency as the original 256-bit CHERI implementation. We extended CHERI LLVM to support CHERI-128 and CHERI-64 capability formats to run multiple benchmarks and applications under both a custom embedded OS and FreeBSD with CHERI support. CHERI-64 and CHERI-128 provide a convincing performance improvement over CHERI-256, reducing L2 cache misses by up to 75% for pointer-heavy benchmarks and greatly reducing the performance overhead versus unprotected MIPS programs. Finally, formal proofs provide assurance that our aggressive arithmetic optimizations do not compromise the correctness of our capability implementation.

In conclusion, this work presents a major maturation of the state of the art in implementing capability fat pointers, and prepares the way for commercial adoption to harden systems against security challenges to computer systems.

13 Acknowledgments

This work is part of the CTSRD, ECATS, and CIFV projects sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237, HR0011-18-C-0016, and FA8650-18-C-7809. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government. Approved for Public Release, Distribution Unlimited. We also acknowledge the EPSRC REMS Programme Grant [EP/K008528/1], the EPSRC Impact Acceleration Account [EP/K503757/1], Arm Limited, and Google, Inc. We would also like to acknowledge Alex Richardson, Lawrence Esswood, Peter Rugg, Peter Sewell, Graeme Barnes, and Bradley Smith who assisted in various capacities to complete this work.

REFERENCES

[1] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *Proceedings of the 41st International Symposium on Computer Architecture*, June 2014.

[2] 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.

[3] 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 Conference on Computer and Communications Security*. ACM, November 2013.

[4] E. Witchel, J. Cates, and K. Asanović, "Mondrian memory protection," *ACM SIGPLAN Notices*, vol. 37, no. 10, pp. 304–316, 2002.

[5] 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.

[6] 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.

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

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

[9] 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.

[10] "BiiN CPU architecture reference manual," BiiN, Hillsboro, Oregon, Tech. Rep., July 1988.

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

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

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

[14] J. Brown, J. Grossman, A. Huang, and T. F. Knight Jr, "A capability representation with embedded address and nearly-exact object bounds," Project Aries Technical Memo 5, http://www.ai. mit. edu/projects/aries/Documents/Memos/ARIES-05. pdf, Tech. Rep., 2000.

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

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

[17] 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*. IEEE, 2004.

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

[19] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti, "Control-flow integrity: Principles, implementations, and applications," in *Proceedings of the 12th ACM conference on Computer and Communications Security*. ACM, 2005.

[20] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, ser. WWC '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14. [Online]. Available: http://dx.doi.org/10.1109/WWC.2001.15

[21] 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.

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

[23] R. Feiertag and P. Neumann, "The foundations of a Provably Secure Operating System (PSOS)," in *Proceedings of the National Computer Conference*. AFIPS Press, 1979, pp. 329–334.

[24] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson, "A Provably Secure Operating System: The system, its applications, and proofs," Computer Science Laboratory, SRI International, Tech. Rep., May 1980, 2nd edition, Report CSL-116.

[25] E. J. Koldinger, J. S. Chase, and S. J. Eggers, "Architecture support for single address space operating systems," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS V. New York, NY, USA: ACM, 1992, pp. 175–186. [Online]. Available: http://doi.acm.org/10.1145/143365.143508

[26] R. B. Lee, "Precision architecture," *Computer*, vol. 22, no. 1, pp. 78–91, Jan 1989.

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

[28] Intel Plc., "Intel® 64 and IA-32 architectures, software developer's manual, Volume 1: Basic architecture," *Intel Corporation*, December 2015.

[29] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis, "Hardware enforcement of application security policies using tagged memory," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 225–240. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855757

[30] 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.

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

[32] "Oracle's SPARC T7 and SPARC M7 server architecture," Oracle, Tech. Rep., 08 2016.

[33] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 51–66. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855768.1855772

[34] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for C," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 245–258. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542504

[35] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen, "Paricheck: An efficient pointer arithmetic checker for c programs," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '10. New York, NY, USA: ACM, 2010, pp. 145–156. [Online]. Available: http://doi.acm.org/10.1145/1755688.1755707

[36] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '02. Berkeley, CA, USA: USENIX Association, 2002, pp. 275–288. [Online]. Available: http://dl.acm.org/citation.cfm?id=647057.713871

Jonathan Woodruff received an Bachelor's degree in Electrical Engineering at the University of Texas at Austin, Masters and PhD degree in Computer Science at the University of Cambridge. He is a Research Associate at the University of Cambridge Department of Computer Science and Technology. His research interests include instruction-set support for security, microarchitectural optimizations for security features, and FPGA prototyping. He has authored 9 papers.

Alexandre Joannou received a degree of Engineering at Ecole Centrale d'Electronique Paris, a Masters in Computer Architecture at Universite Pierre et Marie Currie and a PhD in Computer Science at the University of Cambridge. He is a Research Associate at the University of Cambridge Department of Computer Science and Technology. His research interests include instruction modeling, FPGA prototyping, and security. He is an author of 3 papers and a member of IEEE.

Hongyan Xia received his Bachelor's degree in Electrical and Computer Engineering at the University of Birmingham, Masters in Computer Science at the University of Cambridge. He is a PhD student at the University of Cambridge Department of Computer Science and Technology. His current research interests include memory safety for embedded systems and secure real-time operating systems.

Anthony Fox received Bachelor's and PhD degree in Computer Science at Swansea University. He is a Principal Security Engineer at Arm Ltd. His research interests include formal models of instruction set architectures, interactive theorem proving, and the formal verification of compilers and machine code. He has authored sixteen papers.

Robert M. Norton received a bachelor's degree and PhD In Computer Science from the University of Cambridge. He is currently a research associate, also at the University of Cambridge. His research interests include architectural support for security features including memory safety and formal semantics of instruction sets. He has authored 3 papers.

Thomas Bauereiss is a research associate at the University of Cambridge working on formal modeling and verification of Instruction Set Architectures. He previously developed techniques for verifying information flow security properties of software systems at DFKI Bremen.

Khilan Gudka received a Bachelor's, Masters and PhD degree in Computer Science at Imperial College London. He is a Research Associate at the University of Cambridge Department of Computer Science and Technology. His research interests include program analysis, compilers, application compartmentalisation and concurrency. He has authored 4 papers.

David Chisnall received a Bachelor's degree and PhD in Computer Science at Swansea University. He is a Researcher at the Microsoft Research Cambridge. His research interests include hardware-language co-design and security.

Brooks Davis is a senior computer scientist at SRI International based in Walla Walla, Washington. His research interests include the operating systems, security, and tools to aid the incremental adoption of new technologies. He received a BS in computer science from Harvey Mudd College. He is the author of numerous papers on the open-source FreeBSD operating system. He is a member of the ACM and the IEEE Computer Society.

Nathaniel Filardo received his Bachelor's degrees in Physics and Computer Science from Carnegie Mellon University and his Masters and PhD degrees in Computer Science from Johns Hopkins University. He is a Research Associate at the University of Cambridge Department of Computer Science and Technology. His research interests include architectural security and static type systems.

A. Theodore Markettos received the MA and MEng degrees in Electrical and Information Sciences, and subsequently a PhD in Computer Science, from the University of Cambridge. He is a Senior Research Associate in the Department of Computer Science and Technology at the University of Cambridge. His research interests include hardware security, notably security architectures and I/O security, FPGA design and electronics manufacturing. He has authored 12 papers on related topics.

Michael Roe is a senior research associate at the University of Cambridge Computer Laboratory. He received a PhD in computer science from Swansea University. His research interests include capability systems, cryptographic protocols, and formal methods.

Peter G. Neumann received AM, SM, and PhD degrees from Harvard, and a Dr rerum naturalium from Darmstadt. He is Chief Scientist of the SRI International Computer Science Lab (a not-for-profit research institution), involved primarily in system trustworthiness – including security, safety, and high assurance. In the computer field since 1953, he has numerous published papers and reports. He is a Fellow of the IEEE, ACM, and AAAS.

Robert N. M. Watson received his BS in Logic and Computation with double major in Computer Science from Carnegie Mellon University, and his PhD in Computer Science from the University of Cambridge. He is a University Senior Lecturer (Associate Professor) at the University of Cambridge Department of Computer Science and Technology. His research interests span computer architecture, compilers, program analysis and transformation, operating systems, networking, and security. He is a member of the ACM.

Simon Moore is a Professor of Computer Engineering in the Computer Architecture group at the University of Cambridge Department of Computer Science and Technology, where he conducts research and teaching in the general area of computer design with particular interests in secure computer architecture. He is a senior member of the IEEE.