CHERIoT: Complete Memory Safety for Embedded Devices

Saar Amar*
saaramar5@gmail.com
Microsoft
Tel Aviv, Israel

David Chisnall*
David.Chisnall@cl.cam.ac.uk
Microsoft
Cambridge, UK

Tony Chen
tonychen@microsoft.com
Microsoft
Redmond, Washington, USA

Nathaniel Wesley Filardo*
nwf20@cam.ac.uk
Microsoft
Cambridge, UK

Ben Laurie
benl@google.com
Google
London, UK

Kunyan Liu*
kunyanliu@microsoft.com
Microsoft
San Diego, California, USA

Robert Norton*
robert.norton@microsoft.com
Microsoft
Cambridge, UK

Simon W. Moore
Simon.Moore@cl.cam.ac.uk
University of Cambridge
Cambridge, UK

Yucong Tao
Yucong.Tao@microsoft.com
Microsoft
Mountain View, California, USA

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

Hongyan Xia†*
Jerryxia32@gmail.com
Arm Ltd.
Cambridge, UK

ABSTRACT

The ubiquity of embedded devices is apparent. The desire for increased functionality and connectivity drives ever larger software stacks, with components from multiple vendors and entities. These stacks *should* be replete with isolation and memory safety technologies, but existing solutions impinge upon development, unit cost, power, scalability, and/or real-time constraints, limiting their adoption and production-grade deployments. As memory safety vulnerabilities mount, the situation is clearly not tenable and a new approach is needed.

To slake this need, we present a novel adaptation of the CHERI capability architecture, co-designed with a green-field, security-centric RTOS. It is scaled for embedded systems, is capable of fine-grained software compartmentalization, and provides affordances for full inter-compartment memory safety. We highlight central design decisions and offloads and summarize how our prototype RTOS uses these to enable memory-safe, compartmentalized applications. Unlike many state-of-the-art schemes, our solution deterministically (not probabilistically) eliminates memory safety vulnerabilities while maintaining source-level compatibility. We characterize the power, performance, and area microarchitectural impacts, run microbenchmarks of key facilities, and exhibit the practicality of an end-to-end IoT application. The implementation shows that full memory safety for compartmentalized embedded systems is achievable without violating resource constraints or real-time guarantees, and that hardware assists need not be expensive, intrusive, or power-hungry.

ACM Reference Format:
Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo, Ben Laurie, Kunyan Liu, Robert Norton, Simon W. Moore, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. 2023. CHERIoT: Complete Memory Safety for Embedded Devices. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28–November 01, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3613424.3614266

1 INTRODUCTION

The attack surface of embedded devices is no longer limited to physical attacks, in an increasingly connected world. From consumer electronics (smart watches, WiFi chips) to security-critical devices (self-driving vehicles, aviation and smart grids) and more recently IoT applications, physical isolation is rarely the boundary in modern day embedded devices. With the increase of connectivity comes combinatorial growth of the attack surface. Sadly, the resource constraints and the low-level programming environment mean solving even the most basic problem of memory safety still poses as a monumental challenge. Worse, the gap between the attack surface area and the level of defense widens further when such embedded devices are deployed into complicated multi-tasking scenarios with a Real-Time Operating System (RTOS) and multiple software stacks from different vendors.

Even though researchers have disclosed an alarming number of memory vulnerabilities in recent years [6, 11, 15], the lessons learned from desktop and server systems do not directly translate to embedded systems. Page table techniques, sanitizers, dynamic

*These authors made significant contributions to the design and implementation without which the project would not have been possible.
†Work conducted while at Microsoft.

MICRO '23, October 28–November 01, 2023, Toronto, ON, Canada

ACM ISBN 979-8-4007-0329-4/23/10.
https://doi.org/10.1145/3613424.3614266

instrumentation, higher-level languages and so forth all violate at least one of the real-time, code size, power and compatibility requirements. Instead, the industry has taken an incremental, largely reactive, approach: architectural extensions are modest, narrowly scoped, and heavily constrained by compatibility concerns. These extensions often lack in scalability, generality and language-level tooling, forcing developers back to manual software assertions and code analysis tools, which show limited effectiveness.

Such an astounding security gap must be bridged before we push for even more sophisticated networks among embedded devices. In this paper, we start from the CHERI (Capability Hardware Enhanced RISC Instructions) [27] Instruction Set Architecture (ISA) and investigate its applicability and limitations in resource-constrained scenarios. To address shortcomings of prior attempted adaptations, we co-design a new compartment isolation software model with novel architectural extensions, capability encodings, and micro-architectural accelerations. In tandem, these features, combined with a trusted compartment switcher software routine and partially-trusted memory allocator and scheduler compartments, guarantee complete, deterministic inter-compartment memory safety. Such a strong guarantee comes at a reasonable performance cost while not sacrificing C/C++ source-code level compatibility and real-timeness of the system.

The contributions of this paper are:

(1) An embedded-systems architecture designed to support a (co-designed) RTOS offering complete and deterministic compartmentalized memory safety.
(2) Architectural extensions and hardware acceleration for temporal memory safety of cross-compartment references and the RTOS's shared heap allocator.
(3) Performance evaluation of two embedded cores with different design tradeoffs.
(4) Evaluation of area, power and critical-path of hardware assists for a production-quality core.

2 BACKGROUND

We quickly review compartmentalization, memory safety, CHERI, and how the CHERIoT software stack uses these mechanisms before discussing the novel aspects of our architecture. This whirlwind tour is requisite, as CHERIoT is thoroughly co-designed, above and below the traditional architectural boundary. Very few of its interesting aspects stand alone; most act, at least somewhat, in concert. We believe this is the only way to achieve efficient robustness against a strong threat model, instead of being a temporary setback to attackers and/or useful only for debugging. An overview of the RTOS is available [3] as is the full source code [1].

The core design principles for the CHERIoT platform are:

- Support the *principle of least privilege*, down to fine-grained permissions on individual objects (or fields of those objects) within the system.
- Support the *principle of intentional use*, down to ensuring that individual memory accesses may happen only when presenting a pointer that authorizes the specific operation.
- Provide abstractions that can be surfaced directly in C-like languages, for example protecting objects, not pages, and communicating via function calls between compartments, not marshaled messages, at the lowest levels.
- Avoid requiring any structures in hardware that would introduce nondeterministic latency, for example by requiring caches for hot paths and having slow paths for cache misses as with a conventional MMU and page-table walker.
- Avoid requiring any structures in hardware that would significantly increase the area or power consumption to the degree that it would significantly impact cost or applicable target domains, such as large associative lookups in an MPU or TLB.

2.1 Real-time requirements

A real-time system is one in which the latency of operations is bounded and can be reasoned about. For some applications, those bounds must be very low. From a hardware perspective, the latency of operations should not depend on the data being processed and *must* not depend on other bits of system state. Cores aimed at real-time applications cannot, for example, provide virtual memory that requires traversing page tables on TLB miss because this introduces nondeterminism in memory latency that can be impossible to reason about. Similarly, such systems typically eschew caches.

The CHERIoT system is designed such that none of the hardware operations have nondeterministic latency (though, as a microarchitectural optimization, some may have small variation in cycle time). In addition, we provide extensions that allow software to enforce which code may run with interrupts disabled, which makes it tractable to reason about worst-case latency even in the presence of components provided by mutually distrusting suppliers.

2.2 Software Compartmentalization

Compartmentalization refers to an engineering practice of partitioning a system with the aim of limiting the propagation of damage or malfunction. For software, a compartment is, at least, a collection of code and data, some of which is meant to be private. Compartmentalization then often manifests as "mutual distrust": every compartment considers other compartments, and the surrounding world more generally, to be potentially malicious. The source of this potential malice takes many forms, which includes the inevitability of bugs, untrusted software origins, low quality pre-bundled drivers from toolkits, software supply chain attacks, etc. Other compartments are assumed not just to contain possible bugs but also to actively attack, possibly collaboratively, other compartments in any way possible.

In order for compartments to be useful, however, they must interact with each other and/or the outside world, trustworthy or not. Towards that end, compartments also declare *exports*: procedures and/or (references to) data deliberately offered to the broader world. The data within a compartment may include *imports*: references to exports from other compartments. Even for related compartments (say, A importing B's export), it must be possible to limit the consequences of such relations (continuing the example, A is not licensed to any other part of B, even other exports, that it did not import).

A compartmentalized system will have universally trusted components (the "Trusted Computing Base" or TCB) that enforce the

isolation of, and mediate the controlled sharing between, compartments. These are high-value targets, able to jeopardize aspects of correctness for any or all parts of the system, and, so, should be minimized and carefully audited.[1] In practice, the TCB is a combination of (micro-)architecture, hardware offloads, and software. The (micro-)architecture provides omnipresent, local invariants: things true on a per-instruction basis. Trusted software constructs and enforces global invariants: properties impractical to check in hardware. Offloads sit somewhere in the middle, making a series of localized changes in the service of these global invariants. We shall see that each aspect plays an important role in CHERIoT security.

Our CHERIoT RTOS offers a compartmentalization model of code and data and a traditional multi-threaded time-sharing model of execution. Its threads and compartments are orthogonal. At any time, the processor is running one thread in one compartment and has access to that compartment's code and data memory and that thread's stack memory and register file. *Multitasking scheduling facilities* allow the core to change threads, and *cross-compartment procedure calls* and returns cause it to change compartments.

Compartmentalization is a critical security technology because it protects against unknown attacks by limiting the blast radius of a compromise. Even formally verified systems can contain security vulnerabilities as a result of incomplete specifications or flaws in the underlying axioms. Safe language code can have security bugs from compiler bugs or the interactions with systems (such as most forms of I/O) that are outside of the language's abstract machine.

Even with safe languages, components may come from mutually distrusting sources or with different regulatory requirements. For example, a safety-critical component may need to guarantee that no other software on the system can interfere with its operation. At the same time, the network stack for connecting an IoT device to a back-end service may need to protect TLS client keys and similar from bugs in the rest of the system. In today's systems, this compartmentalization is often achieved by providing multiple microcontrollers, each with separate SRAM, which both adds to the cost and limits flexibility: adding a new isolated concern requires building a new device or weakening security.

2.3 Memory Safety

We define memory safety relative to a compartmentalization model built on mutual distrust. In this model, beyond static sharing by export and import, objects may be *dynamically* shared between mutually distrusting compartments merely by passing a pointer as part of a cross-compartment call. A compartment may contain code written in any language (including assembly, which does not provide an object model), and so memory safety as enforced by the RTOS and architecture is restricted to the cross-compartment case. Each compartment must have the tools that it needs to guarantee that no other compartment may violate its object abstractions. For example, for any object owned by compartment A, compartment B must not be able to: ① Access that object unless passed a pointer to it. ② Access outside the bounds of the object given a valid pointer to that object. ③ Access the object (or the memory that was formerly used for the object) after the object has been freed. ④ Hold a pointer to an object with automatic storage duration ('on-stack' object) after the end of the call in which it was created. ⑤ Hold a temporarily delegated pointer beyond a single call. ⑥ Modify an object passed via immutable reference. ⑦ Modify any object reachable from an object that is passed as a deeply immutable reference. ⑧ Tamper with an object passed via opaque reference. Compartmentalized memory safety is guaranteed regardless of whether the object (or mere memory) is in static data, on the stack, or within the heap.

Additionally, compartments may use the same facilities to achieve defense in depth against bugs *within* themselves. For example, all code in a compartment is permitted to access all of its globals (simply by naming them) but our C/C++ compiler can enforce its object model, even when pointers to globals are exposed to C/C++ code, and so protects against bounds errors even on private globals.

In our software stack, the heap allocator (Section 5.1) is a separate compartment and all guarantees related to heap objects hold for all code outside of said allocator. It is impossible to forge a pointer to a heap object, use out-of-bounds accesses to jump from one heap object to another, or use a heap object after it has been freed.

2.4 Capability Systems and CHERI

A capability is an unforgeable token, which, when presented, can be taken as incontestable proof that the presenter is authorized to have some specified access to the object named in the token [23]. Capability systems define an architectural protection model using graphs of objects (including agents) and the capabilities they hold; the model is largely dual to that of access control lists.[2]

A CHERI memory capability is a particular kind of architecturally guarded fat pointer. Each capability is an integer memory address augmented with *bounds* and *permissions* (read, write, execute, etc.) as well as an out-of-band validity tag bit to enforce its integrity and unforgeability. A CHERI ISA then enforces that each memory access is authorized via a valid capability in a register: the target address must be within bounds and the operation must be permitted. Avoiding an associative lookup, the specific register holding the authorizing capability must be cited by the instruction stream, just as most of today's ISAs cite integer addresses in registers.

Unlike architectural protection mechanisms like segmented memory or Memory Management Unit (MMU) protection tables, which rely on software-managed *indirection tables*, CHERI capabilities are values that flow through the system. The CHERI ISA offers guarded manipulation instructions for capabilities, allowing the construction of less-privileged capabilities from more-privileged ones, but not the reverse. In summary: the bounds may be narrowed, but neither widened nor displaced; permissions may be shed but not regained; and tag bits may be cleared but never set.

2.5 Compartmentalization With CHERI

CHERI capabilities overtly address several concerns of compartmentalized memory safety: ① If software sets the bounds of a pointer

[1] "Trusted" in TCB should be read with resignation and aspiration, not as a pronouncement of fitness for purpose. Merely moving code into a box labeled "TCB" does not improve system security.

[2] Capability systems can be thought of as being "row-wise" representations of systems' abstract access control matrix [16]. Their representations of authority – capabilities – are located within the acting (subject) entities and serve to name the acted-upon (object) entities. By contrast, systems built around access control lists (ACLs) are "column-wise" representations, locating authority with the acted-upon entities with ACL entries naming the acting entities.

to an object, no subsequent action on that pointer will access any adjacent object; similarly, removing write permission renders a pointer permanently read-only. These actions are *per-pointer* and allow software to enforce its (sub)object model's boundaries. ② Absent a capability to an object, software cannot access it, even if it knows the address. In fact, for a CHERI program, the accessible register file defines the root set of capabilities. The program's total authority is completely captured by this set and those that can be (transitively) loaded through them. A program is free to reduce its authority, by deriving suitably narrowed capabilities and erasing the progenitors. Subsequent computation can then act arbitrarily only on the selected subset of resources (to which capabilities exist).

CHERI also adds affordances for "non-monotonic transfers of control", allowing a program to restore (part of) its earlier authority while atomically transferring control to a pre-arranged point in the code.[3] As is only sensible, these transfers are themselves managed by capabilities. A capability to perform such a transfer is opaque, in that its bearer may not load additional capabilities through it, and exercising its authority implies relinquishing program control. More generally, CHERI provides a 'sealing' mechanism for constructing opaque capabilities, which may later be 'unsealed'; again, both actions are authorized by capabilities [27].

Extending CHERI to address the more subtle points of compartmentalization, specifically those concerning *deep* immutability and *temporal* notions like object lifetime and delimited sharing, constitutes the bulk of our effort. These extensions are the key innovations of CHERIoT's capability system and enable its RTOS to offer its complete compartmentalized memory safety.

2.6 RTOS Implementation

Before finally turning to our architecture in detail, it is worth briefly summarizing how our RTOS uses the architecture to make the model concrete. CHERIoT's RTOS defines a compartment to be a contiguous region of code and intra-compartment global data. Compartments' global data may include imports of other compartments' designated exports. Compartments, possibly provided by multiple and mutually-distrusting parties, are statically linked together into a single system image; imports of exports are resolved at this time.

At run-time, threads begin within particular compartments (at their designated entry points). While running within a compartment, absent any intra-compartment special handling, the program counter *capability* grants access to all of the compartment's code. Similarly, an ABI-reserved register, the *globals pointer*, holds a capability granting access to all of the compartment's data. RTOS primitives, totaling a little over 300 hand-written instructions, enforce calling into and returning from compartment entry points, as well as preemptive multitasking, with proper switching of compartment contexts.

CHERIoT also inherits from CHERI its 1-bit information flow control scheme, which classifies capabilities as "global" or "local". Capabilities may transition from global to local, but not the reverse, and storing a local capability requires that the *authority* bear the Store Local permission. The RTOS marks all stack pointers local and ensures that only stack pointers permit storing of local capabilities. This prevents references to the stack from being captured in globals or heap memory, which ① permits stack reuse across compartments, and ② enables *ephemeral delegation* of capabilities by marking them local. (See Section 5.2.)

[3] An imperfect analogy can be made to architectures with protection rings and banked registers. Therein, a more-privileged ring may configure the architecture to create a designated entry point for entry from less-privilege rings and may use its banked view of registers to hold (pointers to) sensitive state inaccessible to the lesser ring(s).

	Permission	Applied to	Permits
GL	Global	Load / Store value	Storing cap. via non-SL cap.
LD	Load data	Load address	Loads (inc. of caps if MC)
SD	Store data	Store address	Stores (inc. of caps if MC)
MC	Memory Cap.	Load / store address	Capability load / store
SL	Store Local	Store address	Stores of non-global caps
LG	Load Global	Load address	Loads of caps with GL LG
LM	Load Mutable	Load address	Loads of caps with SD, LM
EX	Execute	Jump targets	Instruction fetch
SR	System Regs.	Program Counter	Access to special registers
SE	Seal	`cseal` authority	Sealing with given otype
US	Unseal	`cunseal` authority	Unsealing with given otype
U0	User perm. 0	-	For software use

Table 1: Summary of capability permissions.

3 THE CHERIOT ARCHITECTURE

We now turn our attention to our new CHERIoT hardware platform. It revises aspects of the CHERI architecture, improves upon the earlier CHERI-64 capability encoding [29], and incorporates CPU extensions that are foundational for full memory safety.

3.1 Novel features in CHERIoT

The most straightforward changes in CHERIoT are to architectural aspects of CHERI, tailoring for our software model. We focus first on *model* changes, leaving changes to capability *representation* to Section 3.2.

3.1.1 Tailored Capability Permissions. CHERIoT heavily revises the ontology of permissions found within CHERI capabilities. These changes are driven from two different, conflicting needs: our software model requires new expressiveness and yet we must minimize the number of bits used within a capability. We begin by *removing* some unused expressiveness from existing CHERI architectures:

- We drop the unused `cinvoke` instruction and permission.
- The separate load-capability and store-capability permissions are combined into one bit, MC, which modifies the existing load and store permissions to permit loads and stores of capabilities. We did not find it useful to be able to have different access permissions for capabilities and data, other than being able to permit data access but prohibit loads and stores of capabilities.
- Capabilities may not simultaneously permit execution and stores, guaranteeing W⊕X at the hardware level (though retaining the ability for JITs to have separate writeable and executable pointers to the same memory).
- We separate the permissions used for sealing from memory related permissions because their bounds and address refer to a distinct namespaces (otypes rather than memory addresses).

The last two points necessitate three different capability *roots* for writable memory, executable memory and sealing. On CPU reset, all three roots are present in registers. Early-boot software is expected to use these to build narrower capabilities around the system before erasing the roots.

Our software model demands two new permissions:

- Recall from section 2.6 that the RTOS uses CHERI's local/-global information flow control to limit off-stack storage of capabilities. We extend this with a new permission, Load Global (LG), that acts *recursively*: capabilities loaded via a capability without LG will have LG cleared and are marked *local*. Thus, one can delegate a capability to the root of a complex data structure and ensure that any capabilities thence loaded can be held only in registers and on the stack. When the callee returns, its stack will be cleared, ensuring that these capabilities are not captured.
- Similarly, we have a Load Mutable (LM) permission that permits read-only sharing by clearing LM and store permissions on loaded capabilities. This feature is present in ARM Morello but has not previously been featured in a CHERI RISC-V architecture.

The full set of CHERIoT permissions is shown in table 1.

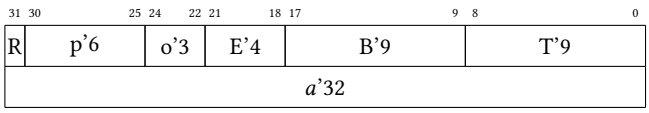
3.1.2 Sentries for Interrupt Control. In an embedded system, particularly one without atomic operations (optional in RISC-V), software often needs to disable interrupts for short periods. In conventional RISC-V, this is accomplished by setting or clearing an interrupt-enabled bit in a control register. In a CHERI system, access to control and status registers (CSRs) is protected by the access-system-registers permission (SR). This permission gives a large degree of control and so we considered separating out the ability to toggle interrupt status into a separate permission.

We realized that, for auditing, it is far more useful to know which code runs with interrupts disabled than it is to know which code may toggle interrupts. CHERI provides a mechanism for guarded control flow: sealed entry ("sentry") capabilities. These are sealed with a specific object type and are unsealed automatically when used as a jump target, but are otherwise unusable. We extended this mechanism to provide *three* sentry types: one enables interrupts, one disables interrupts, and one makes no change to interrupt posture. On a jump-and-link instruction, the link register is written with the sentry type that sets interrupt posture to its current value.[4] This makes it easy to grant a compartment the right to call a particular function with interrupts disabled, without allowing it to arbitrarily disable interrupts (thereby risking system availability).

3.2 CHERIoT capability encoding

Previous CHERI work for embedded devices [2, 29] (see Section 8) directly apply the CHERI Concentrate [26] encoding scheme, from 64-bit systems, to 32-bit addresses, without exploring further optimizations. Notably, the 11-bit permission field, with its orthogonal bits, was retained. As a consequence of this and other inefficient uses of bits, the precision of bounds is significantly reduced. The T and B fields can drop to as low as 3 bits, leading to an average memory fragmentation of $\frac{1}{2^3} = 12.5\%$ for padding and alignment, unacceptable for memory-constrained systems. We now show how CHERIoT addresses these inefficiencies, arriving at an encoding (fig. 1) that optimizes for the typical embedded programming model.

[4]Two more sentry 'otype's are reserved for return-address sentries, one of each interrupt posture. Later revisions of CHERIoT will distinguish forward and backward control-flow arcs.



R reserved bit
p a 6-bit compressed permissions field
o a 3-bit 'object type' ('otype') used to seal capabilities
E a 4-bit exponent used for the bounds encoding
B a 9-bit base used for the bounds encoding
T a 9-bit top used in the bounds encoding
a the 32-bit address of the capability

Figure 1: CHERIoT capability format

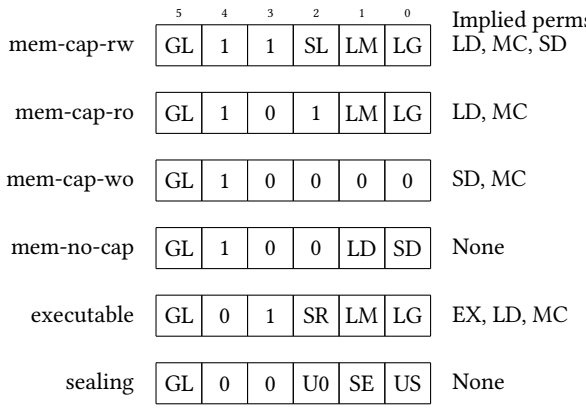


Figure 2: Compressed permission formats

3.2.1 Permission encoding. Most significantly, we introduce *permission* compression. As outlined above (Section 3.1.1), we have identified and removed combinations of permissions that are unwanted. We now exploit the *interdependence* of some permissions to achieve a very compact encoding of our 12 architectural permissions (Table 1) into 6 bits (Figure 2).

We encode the permissions in six different 'formats', with each granting some number of permissions implicitly and encoding the optional permissions that make sense given the implied permissions. This encoding eliminates useless permission combinations. For example, executable capabilities have the implicit permissions required by the ABI for PC-relative addressing, and may optionally grant access to system registers. Capabilities may transition between formats if the permissions are reduced during execution.

As a minor optimisation, we re-ordered the *architectural* view of permissions to place permissions that we anticipate will be most commonly cleared (GL, LG, LM and SD) in the lowest bits. Masks for clearing these may be constructed using a single compressed RISC-V instruction.

3.2.2 Sealing and sentries. CHERIoT reduces the 'otype' field, used to seal a capability, to three bits.[5] We observed in practice that

[5]While this may seem like a severe limitation, given our goal of fine-grained compartmentalization, the RTOS is able to bootstrap a *virtualized* sealing mechanism that, while not identical to CHERI's architectural seals, suffices in all cases we have encountered so far.

$a =$	$a_{\text{top}} = a[31 : e+9]$	$a_{\text{mid}} = a[e+8 : e]$	$a[e-1 : 0]$
$b =$	$a_{\text{top}} + c_{\text{b}}$	B'9	0'e
$t =$	$a_{\text{top}} + c_{\text{t}}$	T'9	0'e

$a_{\text{mid}} < \text{B}$?	T < B ?	c_{b}	c_{t}
no	no	0	0
no	yes	0	1
yes	no	-1	-1
yes	yes	-1	0

Figure 3: CHERIoT bounds decoding.

software does not use the same type for both executable and data capabilities. We enshrine this partition in the encoding, with two *disjoint* sets of 7 'otype' values (0 denotes unsealed), with the set selected by the execute permission. Five of the executable otypes are consumed by (or reserved for) sentries, leaving two for software use. None of the data otypes has significance to hardware; our RTOS allocates four for core components, leaving 3 for other use.

3.2.3 Revised Bounds Encoding. We use a simplified variant of CHERI concentrate to encode the bounds as 2^e-aligned values relative to the address for some exponent, e. Figure 3 shows how the base, b, and top, t, are decoded by inserting B and T at bit e into a and replacing the lower e bits with zeros. The corrections c_{b} and c_{t} account for the possibility of b and t being in different 2^e-aligned regions from a. With this encoding, objects of up to 511 bytes can always be represented precisely, whereas larger objects require their bounds aligned according to the value of e necessary to accommodate their length. To allow the root capabilities to encompass the entire address space an E value of $_{0\times f}$ represents an exponent of 24; other values map directly to their unsigned binary interpretation.

Compared to CHERI concentrate this encoding compromises *representable range* for extra *precision* and reduced complexity. By representable range we mean the range within which the address can move while preserving the same decoded bounds. If the address moves outside of this range the capability is invalidated. C / C++ programs may perform pointer arithmetic that takes the address outside the object bounds, although strictly speaking this is undefined behaviour except in the case of one byte past the end. While the CHERI concentrate encoding goes to considerable lengths to guarantee a minimum representable range beyond the object bounds, the CHERIoT encoding has no such guarantee: in the worst case the representable range is equal to the object bounds, and in all cases addresses below the base are invalid. In the corpus of embedded code that we have compiled so far (including some comparatively large codebases, such as the TPM reference stack and mBedTLS) we have not found this to be a problem. We consider the reduced compatibility an acceptable compromise for the increased precision and reduced complexity. We hypothesise that embedded code is relatively careful about pointer semantics due to being required to execute on more diverse architectures.

Finally, we implemented encoding and decoding in Sail [4] and used its SMT solver backend to check some important properties of the encoding scheme. For brevity, we do not elaborate further here.

All of these optimisations combined give our encoding a *9-bit* precision in the T and B fields. We consider this critical in reducing the average internal memory fragmentation to $\frac{1}{2^9} \approx 0.19\%$, an acceptable cost. In fact, our encoding still has one unused bit available, which could be used for expansion of precision, otypes or new permissions.

3.3 Temporal safety acceleration

The existing CHERI ISA itself does not provide any mechanism for temporal safety. However, prior work [25, 28, 31] has revealed an important insight: that pointer unforgeability and monotonicity offer a foundation for efficient enforcement of temporal safety by means of pointer revocation. The deterministic spatial safety from CHERI hardware enables temporal safety schemes that are also deterministic, that is, references to recycled memory are guaranteed to have been removed prior to reuse. On conventional architectures, in contrast, there is no distinction between pointers and integers and so no guarantee that pointers are derived monotonically. Therefore, temporal safety operating using only conventional architectural mechanisms is probabilistic at best, and the probability can be reduced dramatically by targeted attacks.

Previous CHERI-based temporal safety work has been approaching acceptable performance with reasonable overheads, but uses mechanisms unavailable to embedded systems. Primarily, they implement load and/or store barriers with the MMU, taking advantage of already-incurred overheads and variable latencies within the system. For performance, enforcement of temporal safety is *batched*, with memory 'quarantined' until enforcement finishes; quarantined memory remains accessible to software. This necessitates a weaker security model, differentiating between UAF accesses to quarantined memory and "use after reallocation" accesses to reused memory; only the latter can be guaranteed to be prohibited. Instead, our CPU pipeline offers hardware-assisted temporal memory safety with a stronger security model and without need of an MMU.

3.3.1 Heap revocation bits. As with prior work, we introduce 'revocation' bits to heap allocation granules. Each granule has a corresponding revocation bit, indicating whether this granule belongs to a memory chunk that has been freed and so should not be accessed. We pick 8 bytes as an allocation granule due to capability alignment; this adds SRAM overhead of $\frac{1}{8*8} = 1.56\%$ for each *heap* memory granule. A larger granule size, for a smaller revocation bitmap, is possible, at the cost of some allocations requiring more padding.

We emphasize that this overhead applies only to heap memory. While the simplest approach would be to associate all SRAM with revocation bits, other designs are possible. To name a few design points, the SoC architecture may statically associate only some SRAM with revocation bits, may offer a fixed amount of revocation SRAM to be configurably associated with primary SRAM, or may be able to configurably partition a single SRAM bank into data and revocation regions. Software can ensure that the heap occupies only regions associated with revocation bits and can prefer to place irrevocable resources – code, global data, and thread stacks – in regions without. Thus, the actual SRAM overheads can be much smaller, since heap is only a fraction of the total memory usage of embedded systems, all the way down to none at all if memory is strictly statically allocated.

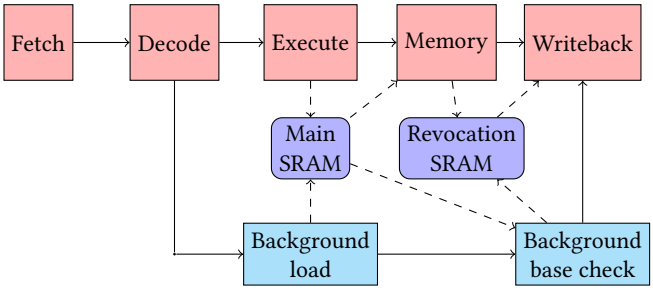


Figure 4: Hardware load filter in a 5-stage pipeline. Arrows indicate pipeline flow whereas dotted arrows indicate SRAM requests and responses.

The revocation bit area is memory-mapped and the RTOS build system and loader ensure that only the heap allocator compartment has access to this region. Upon a `free()` call, the allocator sets the corresponding revocation bits then zeros the freed memory. As with prior work, freed memory is quarantined until the allocator knows there are no outstanding references, at which time the revocation bits are reset and the memory is available for reuse. Unlike prior work, our processor pipeline directly consults these revocation bits!

3.3.2 Hardware load filter. We implement a hardware load filter for all capability load instructions. Upon every `clc $cd, offset($cs)`, the base of the capability *to be moved into*`$cd` is computed and the associated revocation bit is looked up. If the revocation bit is set, this capability points to freed memory and must be invalidated (by clearing at least the tag) before register writeback. This mechanism assumes *spatial safety*: because the allocator has bounded its returned pointer to a particular object, all derived, usable references to that object will have bases within that object.[6]

We exploit the fact that embedded memory is commonly tightly coupled with the CPU, so the revocation bit lookup can be introduced with minimal cycle latency. If an additional read port is dedicated to the revocation bit lookup, the load filter can be implemented without any pipeline stalls in a classic 5-stage pipeline, shown in Figure 4. Here, the capability load instruction initiates a memory read operation in EXE, and gets the response in MEM. The MEM stage computes the base of the loaded capability and initiates a revocation bit read, and gets the revocation bit in WR. WR then strips the tag right before register writeback if the bit is set. The load filter fits in the 5-stage pipeline nicely, as the MEM stage of a CHERI CPU already contains full bounds check logic for certain instructions and finding the base would not be on the critical path.

The load filter is a powerful addition that maintains a crucial invariant: no capabilities that point to freed memory can be loaded into registers. Interestingly, this invariant brings a much simplified version of capability revocation than existing CHERI temporal safety work: sweeping memory to invalidate stale capabilities (capabilities pointing to freed memory with revocation bits set) can be achieved by a simple loop, which loads each capability word and stores it back. Because most embedded CPU pipelines have at least one cycle of load-to-use delay, this loop is unrolled to load two capabilities, avoiding the pipeline bubbles of a straightforward single load and store; complex pipelines may benefit from further loop unrolling. At the end of the loop, all heap memory chunks freed prior to the start of the loop have no stale references to them and can be safely reused.

As the load filter applies to all capability loads in the system, this loop's body must be atomic but the loop may be interrupted or preempted after any iteration. In practice, our software revoker disables interrupts to incrementally sweep parts of memory with a presumably reasonable, and easily changed, batch size. We exploit this preemptable nature of the revoker to allow the allocator to continue servicing requests even while revocation is in progress. Moreover, the RTOS may continue (or begin) a revocation sweep even while the allocator is idle, which may be useful actions if the system has otherwise idled. Towards these ends, the revoker publishes an *epoch* counter, incremented once before starting its sweep of memory and once again upon completion of the sweep. This epoch is crucial for the memory allocator to know whether a revocation pass is underway or finished, as we discuss in Section 5.1.

3.3.3 Background pipelined revoker. The load filter alone is able to significantly simplify sweeping revocation and to make temporal safety possible, with sweeping revocation implemented in software. However, a software implementation consumes CPU cycles in application threads or a dedicated thread. Also, software sweeping must finish before freed memory chunks from the allocator can be returned to free-lists, potentially causing high latency for applications using the heap.

We observe the fact that embedded applications typically spend less than 50% of the CPU performing memory reads and writes, i.e., more than half of the cycles are register-register operations. We implement a "background" engine (fig. 4) which engages the load-store unit whenever the main pipeline is not performing memory operations. The background revoker is a simple state machine, advancing through memory loading and storing each capability word, invalidating capabilities via the load filter. However, a naïve single-stage implementation fails to take full advantage of the memory unit, as the load filter has a one-cycle delay (issue capability word load request, extract base field from the capability word response and issue the revocation bit request, obtain the revocation bit response and then invalidate the capability) to decide whether the capability word should be invalidated. We pipeline the background revocation engine by introducing a second stage; the one-cycle delay is filled by another capability load request, meaning there can be two capability words in flight, achieving maximum throughput.

The background revoker is exposed as an MMIO device with 4 registers: `start`, `end`, `epoch`, and `kick`: `start` and `end` registers specify the region that revoker should sweep. `epoch` is a read-only register directly analogous to the epoch exposed by the software revoker. `kick` register is write-only and writes simply start a revocation pass within [`start`, `end`) with no effect if a revocation pass is already underway.

Race condition with the main pipeline. Notice that the background revocation engine runs right beside application threads and may introduce data races. Consider the following scenario: ① the background revoker loads the word at address A, ② an application thread

[6] More precisely, the use of this mechanism to achieve temporal safety relies on the architectural features of capability tags (to precisely identify pointers) and on monotonicity of capability bounds, as well as on correctness of the bounds set by the allocator software.

writes to A, ③ load filter indicates the capability word points to freed memory, and ④ revoker writes the invalidated word to A. This causes the new word written by the application thread to be overwritten by the revoker, causing logic errors.

To deal with this, store requests from the main pipeline must be made visible to the background revoker as well. The address of the store from the main pipeline must be checked against the addresses of the two capability words in flight in the revoker. If equal, the revoker must reload the word. Note that this is not a problem for a software revoker, where this race condition is avoided by disabling interrupts, so that the load and store will not be interrupted by an application write to the same address. The ability to put mutually distrusting workloads on a single core is one of the benefits of our design but in cases where microcontrollers use multiple cores for performance isolation then the revoker would need to snoop on all memory traffic from either core.

4 IMPLEMENTATIONS

We have built two implementations of the CHERIoT ISA to date. The first is a prototype core based on the BlueSpec Flute core [8], which had already been extended to support CHERI. Flute is a five-stage single-issue in-order pipeline and is written in BlueSpec SystemVerilog. It has a 65-bit (64-bit plus tag) memory bus and a fairly long pipeline by embedded standards. Our second core, the CHERIoT-Ibex, is based on the Ibex core [21], now maintained as part of the OpenTitan project. CHERIoT-Ibex is a small 32-bit core suitable for embedded applications and our version is primarily optimized for area. As with the original Ibex design, it is configurable as either 2-stage or 3-stage in-order pipeline implementation. The CHERIoT-Ibex core's data memory bus is only 33 bits wide. The tag bit is stored in both halves of a capability and, on load, these bits are AND-ed together to give the architectural tag bit. This allows 32-bit data writes to clear the tag bit for the entire capability, without needing to widen the data bus.[7] Other than the data width, the memory interfaces are identical to the original Ibex design.

We select Ibex and Flute to investigate two interesting points in the design space. On one end, Ibex implements a minimal embedded CPU that targets a small silicon area. On the other end, Flute represents a larger CPU that brings higher performance at the cost of higher transistor count. The two CPUs enable us to demonstrate CHERIoT implementation trade-offs under different design requirements and restrictions.

5 MEMORY ALLOCATOR AND COMPARTMENT ISOLATION

Equipped with the hardware assists, the OS and the allocator are now able to enforce full memory safety.

5.1 Heap allocator

We build the heap allocator on top of `dlmalloc`. Boundary tagging and in-band metadata are preferred on embedded devices over size-class or buddy allocators due to memory constraints, and the lack of performance requirements for caches, paged memory or multi-threading throughput. Spatial safety is guaranteed by setting bounds on `malloc()`, excluding the header (which may require small amounts of padding).

We augment `dlmalloc` with separate quarantine lists for temporal safety. Instead of returning memory to the free lists right after `free()`, returned memory chunks are attached to the quarantine list of the current epoch. If the epoch has advanced since the last call to `free()`, the allocator opens a new list. The allocator need track at most only 3 distinct quarantine lists (with different epochs). If a quarantine list has an age of 3 or more (meaning current epoch is at least three greater than the epoch when the list was opened), then chunks on that list must have gone through a revocation sweep.

When enough freed memory has accumulated in quarantine or the system is low on heap memory, the allocator will start the a revocation pass (either using the software revocation loop or by starting the background revocation engine), invalidating all stale capabilities. Since memory chunks are pulled out of a quarantine list and pushed back to free lists for reuse only after a full revocation cycle, it is impossible for allocations to temporally alias, since all stale copies must have been invalidated before a memory chunk is issued to `malloc()` again. Because quarantine is exposed to the architecture and understood by the load filter, we can make the stronger assertion that UAF accesses in general are impossible as soon as `free()` has returned.

5.2 Local-global and stack clearing

Although a heap allocator with spatial and temporal safety provides a generic mechanism, certain function call patterns require passing an object with temporary and scoped lifetime. For example, a caller may invoke a callee and pass on-stack arguments. It is a security risk to allow the callee to hold onto a capability to the caller's stack after the call. A safe allocator can solve the problem by allocating a buffer from the heap and passing the callee a heap buffer. This would add the overhead of a `malloc()` and a `free()` call for every invocation, and increase the frequency of sweeping revocation.

The CHERI ISA provides a crucial mechanism for scoped delegation of objects: the Global (G) and Store-Local (SL) permissions. Capabilities without the G permission are called local capabilities, and can only be stored to memory via a capability that has SL. The details of the RTOS design would not fit in this paper and so we provide only a summary here; a detailed overview and the source code are available [1, 3]. Scoped delegation can be achieved as follows: The bootloader first creates compartments, each of which is defined by a pair of code and global data capabilities. The global pointer of a compartment has its SL permission cleared. On cross-compartment calls, the caller strips the G permission from all objects that it wants to temporarily delegate. The switcher provides the callee with a program stack (by chopping off part of the caller's stack, setting the CHERI bounds in the stack pointer register) that has SL. Note that this stack is the only storage that can store local capabilities, since the code capability is read-only and the global data capability has no SL, therefore arguments from the caller without G cannot be stored outside the stack. When the callee finishes and returns to the caller, the provided stack is zeroed by the kernel, both to avoid leaking secrets and to ensure the references to temporarily delegated objects are destroyed. On return, the caller is confident that the callee no longer has access to those objects.

[7] A similar design is used by IBM's System/38 and later architectures [18, ch. 8].

Scoped delegation exploits the fact that the stack usage of embedded applications is usually limited to a couple of KiBs. The cost of zeroing the stack is bounded by the stack size.

5.2.1 Stack high water mark. When benchmarking the RTOS we observed that the stack clearing necessary on cross compartment calls could have considerable overhead. This is because the compartment switcher does not know how much of the stack has been used prior to the call and so must zero the entire unused portion of the stack before passing it to the callee, and again on return. We therefore devised a simple hardware mechanism that enables the switcher to clear the minimum amount of stack. This consists of two new CSRs protected by the SR permission and accessible only to the switcher: the stack base register and the stack *high water mark*. When starting a thread the switcher sets the stack base register to the lower limit of the thread's allocated stack and the high water mark to its top. On every store the hardware updates the stack high water mark if the store's address is greater than or equal to the stack base and less than the current high water mark. In this way the high water mark tracks the lowest stack address stored to by the current thread and hence the maximum stack usage (note: stacks grow downwards in the RISC-V ABI). On compartment calls and returns the switcher only needs to clear the part of the stack between the high water mark and the current stack pointer and can then reset the high water mark to the current stack pointer. The values of the stack base and high water mark CSRs must be saved and restored on every thread context switch.

This has a low cost in hardware and means stack is never cleared unnecessarily. The amount of stack cleared on entry to a cross compartment call is the difference between the maximum stack usage prior to the call and the stack use at the time of the call (often zero). On return it is exactly that amount of stack used by the callee.

5.3 Unforgeability and monotonicity

The description of the allocator and the stack clearing demonstrates how a deterministic temporal safety scheme must depend on deterministic spatial safety, which, in turn, relies on referential safety, or unforgeability of references. In both the heap and the scoped delegation cases, we guarantee temporal safety on the foundation that a pointer is unforgeable and can only decrease rights. After painting the revocation bits of a memory chunk on `free()`, it is impossible for a stale pointer to escape revocation because CHERI instructions can only increase the base, decrease the top or remove permissions. As a result, a pointer pointing to freed memory must have a base field pointing to revocation bits that have been set, and there is no way for it to escape the revoker.

6 SUMMARY: FULL MEMORY SAFETY

Prior security analysis of CHERI (as used by conventional UNIX-like operating systems) identified limitations in temporal safety [13]. Existing temporal safety work on CHERI, such as Cornucopia [25], does not meet the requirements of low-latency and determinism in embedded systems. Therefore, we explore architectural assists and hardware accelerations to bridge the gap between the ISA and full memory safety (especially temporal safety) in Section 3. First, the hardware load filter ensures it is now possible to enforce temporal safety without stopping the world and without the nondeterminism of an MMU. The background revoker then further accelerates sweeping revocation by taking advantage of spare cycles in the load-store unit, greatly reducing the overhead from a software revoker. We show that such hardware assists need not be intrusive and can fit in the classic 5-stage in-order pipeline nicely. We then describe the necessary allocator and cross-compartment call changes that, together with the hardware assists, enforce complete memory safety.

Unlike prior work, we give a strict and simple definition of complete memory safety. We neither exclude any particular attack surface as a limitation of our work, nor rely on probabilistic approaches that can be circumvented. We believe this is the first embedded system that enforces deterministic spatial and temporal memory safety across compartment boundaries. This is achieved at the ISA and CPU level with a minimum TCB, in contrast to other approaches that require new languages and complete rewrites of the code.

	Ibex 300MHz	
	Gates	**Power** (mW)
RV32E	26988	1.437
RV32E + PMP16	55905 (2.07×)	2.16 (1.50×)
RV32E + capabilities	58110 (2.15×)	2.58 (1.79×)
+ load filter	58431 (2.17×)	2.58 (1.80×)
+ background revoker	61422 (2.28×)	2.73 (1.90×)

Table 2: Area and power costs for variants of Ibex.

7 EVALUATION

7.1 Area, timing and power analysis

To understand area and power costs of CHERIoT, we have implemented several variants of our CHERIoT-Ibex core using TSMC's 28nm HPC+ process [20]. The Ibex variant is the version targeting production use, similar measurements on Flute are not informative because the minimal changes for the prototype left a number of large features (such as supervisor and user modes) that a CHERIoT core does not require. All Ibex configurations had a $F_{\max}$ of 330 MHz. For each, we counted gate equivalents and estimated power draw when running Coremark; results are summarized in table 2.

Broadly speaking, we see that CHERIoT and a 16-way standard RISC-V Physical Memory Protection (PMP) unit, more than double the gates required for Ibex, which is optimized to be small. CHERIoT with its load filter requires an additional 4.5% gate overhead relative to the PMP; adding the optimized background revoker takes the area overhead relative to the 16-element PMP baseline up to a little under 10%. Given the very small baseline, this difference is unlikely to be the deciding factor between a CHERIoT- or PMP-enabled core. Even at 61KGE, the core area is likely to be well under 10% of the area of a cheap IoT SoC. Where area is very tightly constrained but performance is less important, the hardware revoker can be replaced with a software version.

Pre-silicon power evaluation is based on modeling that has a potentially large margin of error and these values should be regarded as preliminary estimates. We believe that much, though not all, of the reported increase in estimated power for variants with CHERIoT is an artifact of the model's over-reliance on *gate count*. In particular, the comparators in the PMP must all be engaged on every load or store, whereas the CHERI version has no equivalent

Configuration	Flute		Ibex	
	Score	Overhead	Score	Overhead
RV32E	2.017	-	2.086	-
+ Capabilities	1.892	5.73%	1.811	13.18%
+ Load filter	1.892	5.73%	1.624	21.28%

Table 3: CoreMark results for our two cores

structures. The hardware background revoker is idle and consumes little power when not in allocation-heavy phases of computation. These preliminary data then suggest that CHERIoT and PMP configurations are likely to have similar power requirements, with CHERIoT perhaps a little higher.

7.2 Performance evaluation

We compile the benchmark using the CHERIoT version of the Clang compiler, which is currently based on Clang 13.0, with -Oz. This flag tells the compiler to optimize for code size, even at the expense of performance and to optimize for performance only where doing so is not expected to impact code size. We choose this option because it is the default for our embedded use cases, where reduced instruction memory has a significant impact on device cost.

The compiler has two known bugs that we did not have time to fix before submission that cause a non-negligible impact on performance and code size, and so these numbers should treated as worst-case. ① The LLVM back end associates arithmetic and folds constants for a large number of address-computation idioms. This does not currently work if the base address is a CHERI capability. This particularly impacts loops that iterate over arrays of structures. ② The LLVM back end currently applies bounds to accesses to globals, even in cases where it can statically prove that the accesses are in bounds. This particularly impacts any phases that access globals. Both of these bugs can be fixed using known techniques and we expect them to be addressed before any CHERIoT silicon is in production.

7.2.1 CoreMark. To evaluate the performance impact of the various features, we first turn to the CoreMark benchmark suite. This suite is intended to measure the core features of a processor. The CoreMark suite runs bare metal and does not depend on our RTOS. The results (CoreMark per MHz) are shown in table 3.

The results show a baseline of the benchmark compiled for RV32E (32-bit address space, 16 registers). The next result is from enabling the CHERI extension, which means that the compiler will use 64-bit capabilities instead of 32-bit integers to represent pointers. This result has the load filter disabled in the cores. The final result shows the overhead of then enabling the load filter.

In both cases enabling capabilities adds some overhead. We believe that much of the overhead on Flute is due to the compiler bugs mentioned earlier, but some is unavoidable. In particular, the compiler must set bounds on stack allocations or address-taken globals. On Ibex, there is additional overhead from the fact that loading or storing a pointer now requires two bus accesses due to the narrower memory bus.

The impact of the load filter is most interesting. On Flute, this is entirely hidden within the existing pipeline structure. Ibex has a shorter pipeline and the extra load-to-use penalty for loads of pointers is noticeable.

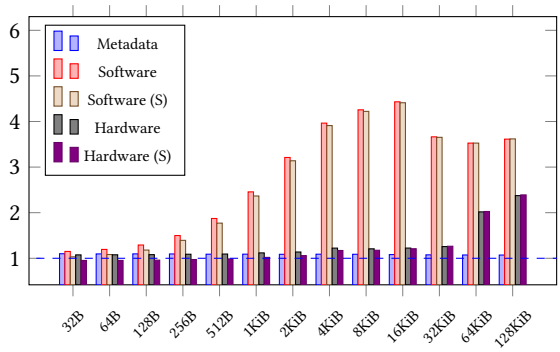


Figure 5: Allocator benchmark results on Flute

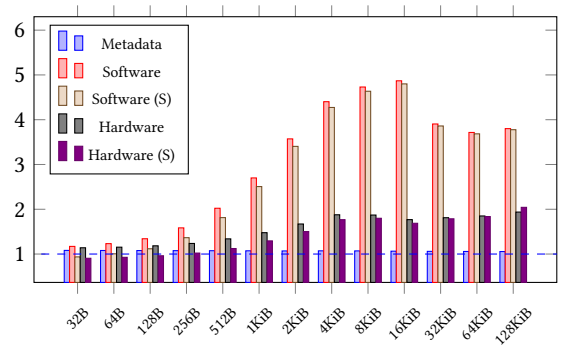


Figure 6: Allocator benchmark results on Ibex

This demonstrates that CHERIoT can be implemented with low performance overhead relative to a RISC-V baseline, when added to a performance-focused embedded core.

7.2.2 Allocation microbenchmark. Having demonstrated the costs of spatial memory safety, we now explore the tradeoff space surrounding the heap. For this evaluation, we have a microbenchmark that allocates and frees a total of 1MiB of memory. We run this benchmark with allocation sizes ranging from 32 bytes to 128 KiB. We run this benchmark in four different configurations: ① With no temporal safety in use at all (**Baseline**),[8] ② with the revocation bits updated, but no sweeping revocation (**Metadata**), ③ with revocation performed in software (**Software**), ④ with revocation offloaded to the hardware (**Hardware**). For each revoker implementation, we also show the results with the stack high water mark (**(S)**) enabled. Raw results are shown in table 4, and fig. 5 and fig. 6 depict overheads relative to the baseline configuration.

The relative cost of scanning all memory for revocation in software initially increases as the allocation size increases, accounting for more than half of the total run time by the time we reach 1KiB

[8]The baseline has no revocation bitmap. Ordinarily, our allocator uses that to detect frees of partial objects, and so the baseline is vulnerable to heap corruption if an attacker frees a pointer to the middle of an allocation. One can build allocators that are robust against this kind of attack without our revocation bitmap, but doing so would incur more memory overhead than the 8 bytes per allocation of our implementation.

	Flute						Ibex					
Size	Baseline	Metadata	Software	Software (S)	Hardware	Hardware (S)	Baseline	Metadata	Software	Software (S)	Hardware	Hardware (S)
32B	$8.77 \cdot 10^7$	$9.64 \cdot 10^7$	$1.01 \cdot 10^8$	$9.05 \cdot 10^7$	$9.43 \cdot 10^7$	$8.36 \cdot 10^7$	$1.26 \cdot 10^8$	$1.37 \cdot 10^8$	$1.48 \cdot 10^8$	$1.18 \cdot 10^8$	$1.44 \cdot 10^8$	$1.14 \cdot 10^8$
64B	$4.57 \cdot 10^7$	$5.01 \cdot 10^7$	$5.46 \cdot 10^7$	$4.95 \cdot 10^7$	$4.92 \cdot 10^7$	$4.36 \cdot 10^7$	$6.53 \cdot 10^7$	$7.05 \cdot 10^7$	$8.05 \cdot 10^7$	$6.56 \cdot 10^7$	$7.53 \cdot 10^7$	$6.05 \cdot 10^7$
128B	$2.32 \cdot 10^7$	$2.54 \cdot 10^7$	$2.99 \cdot 10^7$	$2.74 \cdot 10^7$	$2.50 \cdot 10^7$	$2.22 \cdot 10^7$	$3.32 \cdot 10^7$	$3.58 \cdot 10^7$	$4.45 \cdot 10^7$	$3.71 \cdot 10^7$	$3.93 \cdot 10^7$	$3.19 \cdot 10^7$
256B	$1.19 \cdot 10^7$	$1.30 \cdot 10^7$	$1.79 \cdot 10^7$	$1.66 \cdot 10^7$	$1.30 \cdot 10^7$	$1.16 \cdot 10^7$	$1.71 \cdot 10^7$	$1.84 \cdot 10^7$	$2.71 \cdot 10^7$	$2.34 \cdot 10^7$	$2.12 \cdot 10^7$	$1.75 \cdot 10^7$
512B	$6.33 \cdot 10^6$	$6.90 \cdot 10^6$	$1.19 \cdot 10^7$	$1.12 \cdot 10^7$	$6.92 \cdot 10^6$	$6.25 \cdot 10^6$	$9.08 \cdot 10^6$	$9.76 \cdot 10^6$	$1.84 \cdot 10^7$	$1.65 \cdot 10^7$	$1.22 \cdot 10^7$	$1.02 \cdot 10^7$
1KiB	$3.52 \cdot 10^6$	$3.84 \cdot 10^6$	$8.65 \cdot 10^6$	$8.33 \cdot 10^6$	$3.94 \cdot 10^6$	$3.59 \cdot 10^6$	$5.07 \cdot 10^6$	$5.43 \cdot 10^6$	$1.37 \cdot 10^7$	$1.27 \cdot 10^7$	$7.48 \cdot 10^6$	$6.55 \cdot 10^6$
2KiB	$2.12 \cdot 10^6$	$2.30 \cdot 10^6$	$6.80 \cdot 10^6$	$6.64 \cdot 10^6$	$2.41 \cdot 10^6$	$2.24 \cdot 10^6$	$3.06 \cdot 10^6$	$3.27 \cdot 10^6$	$1.09 \cdot 10^7$	$1.04 \cdot 10^7$	$5.11 \cdot 10^6$	$4.59 \cdot 10^6$
4KiB	$1.47 \cdot 10^6$	$1.61 \cdot 10^6$	$5.84 \cdot 10^6$	$5.76 \cdot 10^6$	$1.80 \cdot 10^6$	$1.73 \cdot 10^6$	$2.13 \cdot 10^6$	$2.28 \cdot 10^6$	$9.38 \cdot 10^6$	$9.10 \cdot 10^6$	$4.00 \cdot 10^6$	$3.77 \cdot 10^6$
8KiB	$1.12 \cdot 10^6$	$1.22 \cdot 10^6$	$4.77 \cdot 10^6$	$4.73 \cdot 10^6$	$1.35 \cdot 10^6$	$1.32 \cdot 10^6$	$1.62 \cdot 10^6$	$1.74 \cdot 10^6$	$7.67 \cdot 10^6$	$7.52 \cdot 10^6$	$3.04 \cdot 10^6$	$2.92 \cdot 10^6$
16KiB	$9.13 \cdot 10^5$	$9.87 \cdot 10^5$	$4.05 \cdot 10^6$	$4.03 \cdot 10^6$	$1.12 \cdot 10^6$	$1.10 \cdot 10^6$	$1.34 \cdot 10^6$	$1.42 \cdot 10^6$	$6.51 \cdot 10^6$	$6.42 \cdot 10^6$	$2.36 \cdot 10^6$	$2.25 \cdot 10^6$
32KiB	$8.10 \cdot 10^5$	$8.70 \cdot 10^5$	$2.97 \cdot 10^6$	$2.96 \cdot 10^6$	$1.02 \cdot 10^6$	$1.02 \cdot 10^6$	$1.19 \cdot 10^6$	$1.26 \cdot 10^6$	$4.66 \cdot 10^6$	$4.61 \cdot 10^6$	$2.16 \cdot 10^6$	$2.13 \cdot 10^6$
64KiB	$7.57 \cdot 10^5$	$8.12 \cdot 10^5$	$2.67 \cdot 10^6$	$2.67 \cdot 10^6$	$1.53 \cdot 10^6$	$1.53 \cdot 10^6$	$1.12 \cdot 10^6$	$1.19 \cdot 10^6$	$4.17 \cdot 10^6$	$4.13 \cdot 10^6$	$2.08 \cdot 10^6$	$2.06 \cdot 10^6$
128KiB	$7.32 \cdot 10^5$	$7.84 \cdot 10^5$	$2.65 \cdot 10^6$	$2.65 \cdot 10^6$	$1.74 \cdot 10^6$	$1.75 \cdot 10^6$	$1.09 \cdot 10^6$	$1.15 \cdot 10^6$	$4.13 \cdot 10^6$	$4.11 \cdot 10^6$	$2.11 \cdot 10^6$	$2.22 \cdot 10^6$

Table 4: The number of cycles taken to allocate 1 MiB of heap memory at different sizes.

allocations. This is because each doubling of the allocation size halves the number of cross-compartment calls to the allocator and so the baseline costs reduce. This happens slightly earlier for Ibex (around 512B allocations) because of the narrower memory bus.

For very large sizes, the cost of revocation dominates the total time. With 128KiB allocations, we must perform a complete revocation sweep for every allocation. The cost of scanning almost 256KiB of SRAM is considerably more than the cost of the allocation, but is more efficient if performed by the hardware revoker.

For small sizes, the cost of cross-compartment calls dominates. The stack high water mark, as described in section 5.2.1, significantly reduces this overhead. For small allocations, this reduces the total cost by 10%, though this benefit decreases as the cross-compartment call cost becomes a smaller part of the total. On Ibex, where the narrower bus makes the cost of zeroing proportionately higher, this brings the cost of spatial and temporal safety, even with revocation performed in software, below the baseline (spatial safety only) for 32- and 64-byte allocations. The combination of both the hardware revoker and the stack high water mark gives better performance than the baseline for allocation sizes up to 512B on Flute (the vast majority of allocations on embedded systems) and close to the baseline on Ibex. This is because the cost of revocation with the hardware revoker is so small that the speedup from the water mark dominates.

Performing revocation in software has a higher overhead (though this would be a lower percentage of the total if the memory is used for anything) and is acceptable for a number of embedded workloads. We believe the overhead from temporal safety with the hardware offload will be sufficiently low to encourage more heap use and reduce the SRAM required for static carve outs in embedded development.

The software revoker performs one load and one store per capability-sized memory location. On Ibex, this becomes four accesses to the main SRAM in total. The hardware revoker does not write back memory unless the tag bit was cleared by the load. The fact that the architectural tag bit is the result of AND-ing together the two microarchitectural tag bits makes two optimizations possible. First, the revoker can perform a single write to invalidate the capability. Second, the revoker can skip the second load if the tag bit is zero in the first half of a capability. We currently implement the first, but not the second, of these optimizations. Ibex is optimized for area and so the current implementation reuses the load checks in the load-capability logic of the main core.

For Flute, the performance of the hardware revoker appears to get worse in the later benchmarks. Flute is the prototype core and so does not have all of the quality-of-implementation features of the production version. In particular, the Flute revoker does not raise an interrupt on completion and so requires software to periodically poll. As the RTOS wakes up the blocking thread to recheck, it performs a flurry of memory accesses, which take precedence over the revoker's access and slow it down.

Somewhat surprisingly, the final case (allocations that are large enough to trigger a revocation each sweep time) becomes slower on Ibex with the stack high-water mark in addition to the revoker. This benchmark variant spends most of its time waiting for the revoker to complete and so the extra two registers that must be saved and restored on thread context switching is visible.

7.2.3 End-to-end performance. To evaluate the end-to-end performance of our system, we assembled an example IoT application. This uses a compartmentalized network stack with the FreeRTOS TCP/IP stack, mBedTLS, and the FreeRTOS MQTT library, each in a separate compartment. These connect to the Azure IoT Hub and fetch some JavaScript bytecode, which is then run using the Microvium JavaScript interpreter (in a separate compartment). Every network packet that is sent and received is a separate heap allocation, protected by temporal safety, as are the chunks of memory that make up the JavaScript heap. Microvium does not reuse memory between garbage collection passes and so our temporal safety guarantees also hold for JavaScript objects accessed from C code. The JavaScript is invoked every 10ms to animate the LEDs on the FPGA dev board, which is running the CHERIoT Ibex at 20MHz. The CPU load, averaged over a minute of the benchmark run (so including the time taken to establish the TLS connection) is 17.5% (i.e. 82.5% of the total CPU time is allocated to the idle thread). We believe that this demonstrates that even an area-optimized core at a low clock speed can meet the requirements of a large number of embedded use cases, without the need to rewrite the source code (the majority of this example is running existing open-source embedded components).

8 RELATED WORK

MPU/PMP and TrustZone-M. Embedded systems generally lack the protection features (segmentation, paging) of larger systems and may instead have a *memory protection unit* (MPU) [19]. (RISC-V calls their variant a *Physical Memory Protection* unit [24, §3.7].) An MPU applies region-based policies to physical memory by performing associative lookup of each request's address against each configured region. This lookup places intense downward pressure on the number of regions that may be configured at once. Software may have to compensate by dynamically managing MPU configuration.

Extensions to MPUs, such as in TrustLite [14] and TyTAN [9], attempt to reduce the context switch overhead by making protection table rows further condition on the instruction pointer. This comes at the cost of fewer rows applicable to any one compartment. The software model remains one of manually managed shared memory regions.

Arm's TrustZone-M [30] adds MPU-like mechanisms ("Attribution Unit"s) to partition memory between two 'worlds', dubbed 'secure' and 'insecure'. Each world has two privilege rings with traditional behavior, and new cross-world call mechanisms are added, with secure world entry restricted to designated entrypoints. The software model is one of lopsided distrust: the secure world may access the insecure world, but not vice versa. This risks placing the *entirety* of the secure world in the TCB for the untrusted world. TrustZone-M offers no new mechanisms for memory safety, and its compartmentalization granularity remains essentially unaltered from MPU-based designs.

Virtual Memory. Embedded systems forgo MMUs because performance requires a sizable and fast cache (the Translation Lookaside Buffer) and/or due to the non-deterministic and complex costs of paging entry lookup. Despite that, the flexibility of MMUs' paged address spaces remains alluring, *especially* if a system is shared with soft- or non-real-time tasks, where the latter concerns no longer apply. With some effort, one can treat real-world architectures' TLBs much like PMPs, albeit with address remapping, and obtain behavior compatible with real-time tasks [7]. The analogy to PMPs also largely applies to the resulting software model. In contrast, outside using its heap allocator (a concept generally denied to hard-real-time tasks anyway), CHERIoT's protections are intrinsically deterministic without sacrificing scalability.

Embedded CHERI. To our knowledge, there have been two prior attempts at scaling CHERI, (co-)designed with UNIX-like systems in mind, down to embedded systems.

The first, CheriRTOS [29], introduced the first 64-bit CHERI capability format (for 32-bit addresses). Its RTOS explored the use of capabilities for accessing a shared heap, but centered around *task*-based compartmentalization, coupling together threads and data. Code within compartments used *integer* offsets (relative to ambient capabilities) to refer to code, global, and stack memory; that is, capabilities were used only to describe, and at the interfaces between, compartments. While it showed expressivity improvements using CHERI rather than an MPU, its software model nevertheless left several important concerns out of scope: ① The shared heap was not temporally safe, allowing Use After Free (UAF) accesses. ② No safe mechanism to share on-stack objects across compartments. ③ Memory safety was opt-in, requiring annotations on shared pointers. ④ Error handling and availability were neglected. As a consequence, (mutually) distrusting compartments communicated by message passing (of bytes, not capabilities), suffering serialization and extra memory copies, and largely forewent use of the shared heap.

A later attempt, CheriFreeRTOS [2], introduced automatic compartmentalization at library boundaries, static policies for cross-compartment communication, and brought *availability* into scope. However, temporal safety was still out of scope: it lacks any heap revocation mechanism or an analogue of our Load Global permission. To truly adhere to mutual distrust, sharing via bounded capabilities is still difficult, and one may still need to resort to message passing to avoid leaking memory references to another compartment.

Software approaches. Most embedded software is written in unsafe languages (without any mitigations) but several projects have attempted to shift this. Some approaches [12, 22] propose subsets of C that can be statically checked to be memory safe. These have clear benefits but do not address the large body of code not written in these styles or subsets, do not address assembly code, and do not provide a foundation for mutual distrust.

More recently, there has been an increased interest in using safe systems languages for embedded development [17]. We welcome this and aim to bring Rust support to CHERIoT soon. Safe languages alone require all code to be compiled with a trusted compiler, which is problematic in some supply-chain scenarios. They also typically require that everything other than core system code avoids the "unsafe" mode of the language that would allow a component to violate the security of another.

Garbage Collection. The CHERIoT load filter is inspired by the "load barrier" mechanisms of garbage collection (GC) [5, 10]. Like us, some GCs need to intercept, and possibly modify, pointers before they are exposed to the application. However, our uses are duals: such GCs ensure that pointed-to objects remain live, while CHERIoT's load filter prevents the application from seeing pointers to dead objects. Because our load filter only requires the load of a revocation bit, we have not also tasked it with patching the in-memory copy of the loaded capability.

ACKNOWLEDGMENTS

Approved for public release; distribution is unlimited. This work was supported in part by Defense Advanced Research Projects Agency (DARPA) Contract No. HR001122C0110 ("ETC") and Innovate UK project Digital Security by Design (DSbD) Technology Platform Prototype, 105694.

9 CONCLUSION

We have demonstrated that a capability-based ISA extension, co-designed with an RTOS and compartmentalization model, can provide fine-grained memory safety along with the building blocks for mutually distrusting compartments to share a heap. We have also shown that this can be implemented in different pipelines with different tradeoffs between performance and complexity. We have also demonstrated that these strong security guarantees, which are significantly in advance of anything currently available, can

be implemented with a modest hardware cost, comparable to the simplest existing security features. In addition, we demonstrated two hardware extensions that provide a significant performance improvement in providing a safe shared heap.

REFERENCES

[1] [n. d.]. The CHERIoT RTOS. https://github.com/Microsoft/CHERIoT-RTOS Accessed November 2023.

[2] Hesham Almatary. 2022. *CHERI compartmentalisation for embedded systems.* Technical Report UCAM-CL-TR-976. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-976

[3] Saar Amar, Tony Chen, David Chisnall, Felix Domke, Nathaniel Filardo, Kunyan Liu, Robert Norton-Wright, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. 2023. *CHERIoT: Rethinking security for low-cost embedded systems.* Technical Report MSR-TR-2023-6. Microsoft. https://www.microsoft.com/en-us/research/uploads/prod/2023/02/cheriot-63e11a4f1e629.pdf

[4] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages.* https://doi.org/10.1145/3290384 Proc. ACM Program. Lang. 3, POPL, Article 71.

[5] Henry G. Baker. 1978. List Processing in Real Time on a Serial Computer. *Commun. ACM* 21, 4 (apr 1978), 280–294. https://doi.org/10.1145/359460.359470

[6] Gal Beniamini. 2017. Over the air: Exploiting Broadcom's Wi-Fi Stack (part 1). https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html

[7] M.D. Bennett and N.C. Audsley. 2001. Predictable and efficient virtual addressing for safety-critical real-time systems. In *Proceedings 13th Euromicro Conference on Real-Time Systems.* 183–190. https://doi.org/10.1109/EMRTS.2001.934028

[8] Bluespec. [n. d.]. Bluespec/Flute: RISC-V CPU, simple 5-stage in-order pipeline, for low-end applications needing MMUs and some performance. https://github.com/bluespec/Flute

[9] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. 2015. TyTAN: Tiny trust anchor for tiny devices. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC).* 1–6.

[10] Cliff Click, Gil Tene, and Michael Wolf. 2005. The Pauseless GC Algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments* (Chicago, IL, USA) *(VEE '05).* Association for Computing Machinery, New York, NY, USA, 46–56. https://doi.org/10.1145/1064979.1064988

[11] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. 2014. A Large-Scale Analysis of the Security of Embedded Firmwares. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (San Diego, CA) *(SEC'14).* USENIX Association, USA, 95–110.

[12] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. 2005. Memory Safety without Garbage Collection for Embedded Applications. *ACM Trans. Embed. Comput. Syst.* 4, 1 (feb 2005), 73–111.

[13] Nicolas Joly, Saif ElShere, and Saar Amar. 2020. *Security analysis of CHERI ISA.* Technical Report. Microsoft Security Response Center. https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20CHERI%20ISA.pdf

[14] Patrick Koeberl, Steffen Schulz, Vijay Varadharajan, and Ahmad-Reza Sadeghi. 2014. TrustLite: A Security Architecture for Tiny Embedded Devices.

[15] laginimaineb. 2016. QSEE privilege escalation vulnerability and exploit. https://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html

[16] Butler W. Lampson. 1974. Protection. *SIGOPS Oper. Syst. Rev.* 8, 1 (jan 1974), 18–24. https://doi.org/10.1145/775265.775268

[17] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17).* Association for Computing Machinery, New York, NY, USA, 234–251.

[18] Henry M. Levy. 1984. *Capability-Based Computer Systems.* Digital Press. https://doi.org/10.1016/C2013-0-01290-X

[19] Arm Limited. 2022. Armv8-M Memory Model and Memory Protection User Guide. (Dec 2022). Issue 0100-01. https://developer.arm.com/documentation/107565/latest

[20] Taiwan Semiconductor Manufacturing Company Limited. [n. d.]. *28nm Technology.* https://www.tsmc.com/english/dedicatedFoundry/technology/logic/l_28nm

[21] lowRISC. [n. d.]. LowRISC/IBEX: Ibex is a small 32 bit RISC-v CPU core, previously known as Zero-riscy. https://github.com/lowRISC/ibex

[22] Daniele Midi, Mathias Payer, and Elisa Bertino. 2017. Memory Safety for Embedded Devices with NesCheck. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (Abu Dhabi, United Arab Emirates) *(ASIA CCS '17).* Association for Computing Machinery, New York, NY, USA, 127–139.

[23] Jerome H. Saltzer and Michael D. Schroeder. 1975. The Protection of Information in Computer Systems. In *Proceedings of the IEEE 63-9.*

[24] Andrew Waterman, Krste Asanović, and John Hauser (Eds.). 2022. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture.* https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

[25] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. 2020. Cornucopia: Temporal Safety for CHERI Heaps. In *2020 IEEE Symposium on Security and Privacy (SP).* 608–625. https://doi.org/10.1109/SP40000.2020.00098

[26] 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, and Simon W. Moore. 2019. CHERI Concentrate: Practical Compressed Capabilities. *IEEE Trans. Comput.* (April 2019). https://www.microsoft.com/en-us/research/publication/cheri-concentrate-practical-compressed-capabilities/

[27] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA) *(ISCA '14).* IEEE Press, 457–468.

[28] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. 2019. CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52).* Association for Computing Machinery, New York, NY, USA, 545–557. https://doi.org/10.1145/3352460.3358288

[29] Hongyan Xia, Jonathan Woodruff, Hadrien Barral, Lawrence Esswood, Alexandre Joannou, Robert Kovacsics, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alexander Richardson, Simon W. Moore, and Robert N. M. Watson. 2018. CheriRTOS: A Capability Model for Embedded Devices. In *2018 IEEE 36th International Conference on Computer Design (ICCD).* 92–99. https://doi.org/10.1109/ICCD.2018.00023

[30] Joseph Yiu. 2015. ARMv8-M Architecture Technical Overview. (Nov 2015). https://community.arm.com/cfs-file/__key/communityserver-blogs-components-weblogfiles/00-00-00-21-42/8461.Whitepaper-_2D00_-ARMv8_2D00_M-Architecture-Technical-Overview.pdf

[31] Tong Zhang, Dongyoon Lee, and Changhee Jung. 2019. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19).* Association for Computing Machinery, New York, NY, USA, 631–644. https://doi.org/10.1145/3297858.3304017