Technical Report

UCAM-CL-TR-953
ISSN 1476-2986

Number 953

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

DSbD CHERI and Morello Capability Essential IP (Version 1)

Robert N. M. Watson, Jonathan Woodruff,
Alexandre Joannou, Simon W. Moore,
Peter Sewell, Arm Limited

December 2020

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This work was funded by the UK Government's Industrial Strategy Challenge Fund (ISCF) under the Digital Security by Design (DSbD) Programme delivered by UK Research and Innovation (UKRI), as part of the DSbD Technology Platform Prototype project (105694).

Approved for public release; distribution is unlimited. Sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237 ("CTSRD"), with additional support from FA8750-11-C-0249 ("MRC2"), HR0011-18-C-0016 ("ECATS"), and FA8650-18-C-7809 ("CIFV") as part of the DARPA CRASH, MRC, and SSITH research programs. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

Additional support was received from St John's College Cambridge, the Google SOAAP Focused Research Award, a Google Chrome University Research Program Award, the RCUK's Horizon Digital Economy Research Hub Grant (EP/G065802/1), the EPSRC REMS Programme Grant (EP/K008528/1), the EPSRC Impact Acceleration Account (EP/K503757/1), the EPSRC IOSEC grant (EP/EP/R012458/1), the ERC Advanced Grant ELVER (789108), the Isaac Newton Trust, the UK Higher Education Innovation Fund (HEIF), Thales E-Security, Microsoft Research Cambridge, Arm Limited, Google DeepMind, HP Enterprise, and a Gates Cambridge Scholarship.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

The CHERI protection model extends contemporary Instruction Set Architectures (ISAs) with support for architectural capabilities. The UKRI Digital Security by Design (DSbD) programme is supporting the creation of Arm's prototype Morello processor, System-on-Chip (SoC), and board. Morello experimentally incorporates the CHERI protection model, developed at the University of Cambridge and SRI International, into the ARMv8-A architecture. This document declares a set of *capability essential IP* – ideas essential to the creation of a contemporary CHERI capability system in architecture and microarchitecture. Arm and Cambridge agree that they have made this IP available for use without restriction. This document also identifies a set of CHERI background documents that may be of value as prior art.

Acknowledgments

The authors gratefully acknowledge the many contributors to, and supporters of, the CHERI and Morello projects at SRI International, the University of Cambridge, Arm, and beyond.

The contributions of our coauthors on the CHERI architecture specification have been essential to the creation of CHERI:

Peter G. Neumann, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Ben Laurie, A. Theodore Markettos, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Stacey Son, and Hongyan Xia.

We further thank our many colleagues at Arm, who have supported this work throughout, including Karthik Muthusamy.

We thank our sponsors UK Research and Innovation (UKRI) and the Industrial Strategy Challenge Fund (ISCF), and especially John Goodacre and Georgios Papadakis, without which the Digital Security by Design programme, and Morello, would not have been possible.

Finally, we are grateful to Howie Shrobe, Robert Laddaga, Stu Wagner, Jonathan Smith, John Launchbury, Dale Waters, Linton Salmon, Keith Rebello, Daniel Adams, Laurisa Goergen, Marnie Dunsmore, John Marsh, and Austin Roach at DARPA for their support throughout the CHERI research.

1 Statement of intent

The objective of this document is to lay out the Intellectual Property (IP) positions of the University of Cambridge and Arm Limited with respect to the Digital Security by Design (DSbD) programme and the Arm Morello architecture, processor, System-on-Chip (SoC), and board. As part of this work, we have identified a set of Capability Essential IP: a set of key architectural concepts required to implement the CHERI protection model in contemporary architecture and microarchitecture, which are elaborated in this document.

Our intent, spelled out in detail in the remainder of this report, is to assert that we have not protected Capability Essential IP, and in as much as is necessary, grant permission for them to be used freely and without restriction in third-party implementations. Our aim is to encourage the widespread adoption of these technologies, which we believe is best done through the avoidance of intellectual property protection or limitations on free use. This report does not make any assertions regarding IP that is not Capability Essential. We cannot, and do not, warrant that the IP described in this document is free from protection by any third parties, or from other restrictions (such as export control).

In Section 2, we define key terms, including Capability Essential IP, and describe our IP position.

In Sections 3 and 4, we enumerate essential architectural and microarchitectural IP.

In Section 5, we provide a bibliography and identify an additional set of potentially relevant background material encapsulated in past University of Cambridge technical reports, papers, and open-source hardware and software releases. These do not define capability essential IP, but may be of interest to parties considering the design and implementation of protection models in architecture and microarchitecture as prior art.

As this is our first attempt to enumerate Capability Essential IP, we expect that there will be future versions of this report expanding on these concepts.

2 IP position and assertion

This report encapsulates our current understanding of *capability essential IP*, the essential concepts underlying the implementation of the CHERI protection model in computer architecture, microarchitecture, and software. These architectural and microarchitectural concepts are expressed explicitly in this document in Sections 3 and 4.

2.1 Capability essential IP

"Capability Architecture" means the microprocessor Instruction Set Architecture (ISA), cache and memory system and exception architecture mechanisms required to support capability-based memory pointer protection and in-address-space compartmentalisation. This includes architectural features required to implement full or compressed fat-pointer style capabilities with base/bounds, permissions, guarded manipulation, object types, information flow-control, safe compression, tags for integrity, capability-based domain switching mechanisms, temporal memory safety, and the capability-based control of Direct Memory Access (DMA), as well as the minimum page table modifications required to support the capability architecture.

"Capability Essential IP" means IPR which, absent any license, is unavoidably infringed to make, use, market, import, offer to sell, or sell, and to otherwise directly or indirectly distribute, or otherwise commercially exploit products which implement a Capability Architecture. "Unavoidably infringed" means that there is no commercially and technically viable alternative way to implement that element of a Capability Architecture without resulting in such infringement. To the extent that such Intellectual Property constitutes a patent, only those claims of such patent which are unavoidably infringed shall be deemed to be Capability Essential IP.

Capability Essential IP without limitation includes Intellectual Property listed in Schedule A and excludes Intellectual Property listed in Schedule B. For the avoidance of doubt, the presence of Intellectual Property in Schedule A does not constitute agreement that, and is not determinative of whether, such Intellectual Property is unavoidably infringed when implementing a Capability Architecture. Capability Essential IP shall not include any Arm Implementation IP or features which are directed to improving the efficiency of a Capability Architecture.

SCHEDULE A

Capability Essential IP includes those architectural features set out in the defined term together with software and software mechanisms such as:

- compiler-based techniques for annotating or mechanically identifying opportunities to use capabilities to represent language-level pointers and references;
- linker and debugger support for capabilities;
- operating-system support such as capability context switching and virtual-memory support for tags, and compartmentalisation support via fast hardware-software domain switching;
- runtime support for creating and maintaining of compartments based on capabilities; and
- other capability features which are fundamentally required to support the portability of software between implementations of a Capability Architecture.

It also includes tests for software features such as compiler support, linker support, and operating-system support for capabilities.

SCHEDULE B

Capability Essential IP excludes:

- technology which is not directly related to capability-based protection mechanisms, such as general-purpose processing technology suitable for use in systems which do not implement a Capability Architecture.
- non-essential microarchitectural optimisations of capability mechanisms, and technology directed to the interaction of non-essential processing system components with capability mechanisms.
- Technology that may be used to develop, design, manufacture, sell or use any product or portion thereof that implements a Capability Architecture but which is not part of such Capability Architecture (examples of such technologies include without limitation electronic design automation technology and semiconductor manufacturing technology).

2.2 IP position

The University of Cambridge and Arm Limited have not filed, and will not file, for patent protection for any Capability Essential IP described in this document.

3 Capability essential architecture

3.1 CHERI

This text is drawn from Section 3.12 of the CHERI ISAv8 specification [Wat20b]:

- Capabilities can be used to implement pointers into virtual address spaces (or physical address spaces for processors without virtual memory, or with virtual memory disabled);
- Tags on registers or in memory determine whether they are valid capabilities for loading, fetching, or jumping to;
- Tagged registers can contain both data and capabilities, allowing (for example) capability oblivious memory copies;
- Tags on capability-sized, capability-aligned units of memory preserve validity (or invalidity) across loads and stores to memory;
- Tags are associated with physical memory locations – i.e., if the same physical memory is mapped at two different virtual addresses, the same tags will be used;
- Attempts to store data (rather than a valid capability) into memory that has one or more valid tags will atomically clear the tags on any affected memory;
- Capability loads and stores to memory offer strong atomicity with respect to capability values and tags preventing race conditions that might yield combinations of different capability values, or the tag remaining set when a corrupted capability is reloaded;
- Capabilities contain bounds and permissions; a capability's address is able to float freely within (and to varying extents, beyond) the bounds;
- Permissions control both data and control-flow operations;
- Guarded manipulation in the architecture (and, implicitly, microarchitecture) implements monotonicity: rights can be reduced (but not increased) through valid manipulations of capabilities;
- Invalid manipulations of capabilities violating guarded-manipulation rules lead to an exception or clearing of the valid tag, whether in a register or in memory, with suitable atomicity;
- Loads via, stores via, and jumps to capabilities are constrained by permissions and bounds, and will throw an exception, clear data, or clear tags on a violation;
- For bounds or other violations on a branch or jump instruction, the exception could be thrown on the source instruction, or when fetching the at the destination address;
- Capability exceptions, in general, are delivered with greater priority than MMU exceptions;
- Permissions on capabilities include the ability to not just control loading and storing of data, but also loading and storing of capabilities;
- Capability-unaware loads, stores, and jump operations via integer pointers are constrained by implied capabilities such as the Default Data Capability and Program Counter Capability, ensuring that legacy code is constrained;
- If present, the Memory Management Unit (MMU), whether through extensions to software managed Translation Look-aside Buffers (TLBs), or via page-table extensions for hardware managed TLBs, contains additional permissions controlling the loading and storing of capabilities;
- That MMU-enforced permissions may clear tags or throw exceptions if violated (possibly as configurable option);
- C-language compatibility is maintained through definitions of NULL to be untagged, zero-filled memory, instructions to convert between capabilities and integer pointers, and instructions providing C-compatible equality operators;

- Reserved capabilities, whether in special registers or within a capability register file, allow a software supervisor to operate with greater rights than non-supervisor code, recovering those rights on exception delivery;
- A capability flow-control model to allow the propagation of capabilities to be constrained, preventing the capabilities marked as local from being stored via capabilities marked to prevent that;
- Sealed capabilities allow a non-monotonic escalation of privilege associated with a constrained control-flow transition to a defined address. Subject to the use of suitable instructions, and appropriate permissions, a pair of sealed capabilities with identical object types allow access to unsealed versions of the capabilities, with code beginning execution at one of them. This enables software-enabled behaviours such as software compartmentalization.
- Sealed entry capabilities likewise allow non-monotonic escalation of privilege associated with a constrained control-flow transition to a defined address. Subject to use of suitable instructions, and appropriate permissions, a single sealed entry (sentry) capability allows code to begin execution via an unsealed version of the same capability.
- By clearing architecture-defined permissions, and utilizing software-defined permissions, capabilities can be used to represent spaces other than the virtual address space;
- For compressed capabilities, addresses can stray well out-of-bounds without becoming unrepresentable;
- For compressed capabilities, alignment requirements do not restrict common object sizes and do not overly restrict large objects beyond common limitations of allocators and virtual memory mapping; and
- That through inductive properties of the instruction set, from the point of CPU reset, via guarded manipulation, and suitable firmware and software management, it is not possible to "forge" capabilities or otherwise escalate privilege other than as described by this model and explicit exercise of privilege (e.g., via saved exception-handler capabilities, unsealing, etc).

3.2 Morello

The following high-level concepts implemented in Morello [Arm20] can be considered capability-essential IP:

- Sealed capabilities allow a non-monotonic escalation of privilege associated with a constrained control-flow transition to a defined address. Subject to the use of suitable instructions, and appropriate permissions, **a sealed capability allows controlled access to two unsealed capabilities in memory with code beginning execution at one of them**. This enables software-enabled behaviours such as software compartmentalization.
- Sealed capabilities allow a non-monotonic escalation of privilege associated with a constrained control-flow transition to a defined address. Subject to the use of suitable instructions, and appropriate permissions, **a sealed capability allows access to an unsealed version of the same capability and controlled access to a further capability in memory with the code beginning execution at that capability**. This enables software-enabled behaviours such as software compartmentalization.
- An identifier managed by software using capability permissions and used by hardware to discriminate speculation contexts. This enables separation of a software defined set of compartments in an environment where uncontrolled speculation can leak information.

4 Capability essential microarchitecture

4.1 CHERI

This text is drawn from Chapter 11 of the CHERI ISAv8 specification [Wat20b]:

The CHERI architecture has been designed to fit into modern RISC pipelines without disturbing existing control-flow or data paths.

As a result, microarchitectural concerns are simpler than they could otherwise be, but there are nevertheless several innovations that have been developed for prototype implementations that may be relevant to specific microarchitectures that implement the CHERI model.

The following repositories hold open-source CHERI implementations or libraries, and are referenced in the sections below.

- [CHERI-MIPS]: Original reference implementation of the CHERI-MIPS ISA set
- [cheri-cap-lib]: A library of reference capability algorithms developed for CHERI-MIPS and adapted for CHERI-RISC-V implementations, including for capability compression
- [TagController]: Tag controller for emulating a tagged memory using a hierarchical in-memory table developed for CHERI-MIPS and adapted for CHERI-RISC-V implementations
- [CHERI-Piccolo]: CHERI-RISC-V CPU with a simple 3-stage pipeline, for low-end applications (e.g., embedded, IoT)
- [CHERI-Flute]: CHERI-RISC-V CPU with a simple 5-stage in-order pipeline, for low-end applications needing MMUs and some performance
- [CHERI-Toooba]: CHERI-RISC-V CPU with a superscalar, out-of-order pipeline and multi-core capable; based on RISCY-OOO from MIT

4.1.1 Capabilities in the pipeline

Capability instructions in the CHERI architecture are modelled after integer operations and are almost entirely single-cycle in the open-source implementations. This makes it possible for a CHERI architecture to unify integer and capability registers and execution paths.

4.1.1.1 Register file

Capabilities may be stored in an extended, integer register file, or may use a separate, dedicated register file.

The CHERI-MIPS architecture and microarchitecture use a separate capability register file, enabling instructions to access capability operands in addition to two integer operands. CHERI-MIPS uses a dedicated module to perform CHERI operations, but this runs in lockstep with the main pipeline as many capability instructions have integer operands or results, and all legacy memory operations implicitly have DDC as a capability operand. This microarchitecture is described in [Woo14a]. A superscalar and out-of-order implementation in this style would dedicate execution units to capability operations with ports into the capability register file which would not be necessary in integer execution units, similar to specializations used for floating point execution units.

Our CHERI-RISC-V architecture and microarchitectures use a unified register file for both integers and capabilities. All integer execution units are extended to implement capability manipulation operations.

It should also be possible to implement a unified register-file architecture with a microarchitecturally split register file. One such option would split the physical register file

between the lower half, which is used by integer operations, and the upper half which is consumed (and produced) exclusively by capability operations. This division could enable specialization of execution units to reduce the cost of capabilities to integer paths. This would also reduce total register file storage, as all integer operands and results would not require an entry in the capability register file, while complicating renaming due to operands being split between two renamed register files.

4.1.1.2 Capability decoding

CHERI capabilities are compressed (see Section 11.2 of [Wat20b]) and various microarchitectures may choose to decode capabilities in stages when there is opportunity in the pipeline. The open-source CHERI implementations, including CHERI-MIPS, Piccolo, Flute, and Toooba, use 3 stages of decompression. The first is the fully compressed, architectural in-memory format. The second is a lightly decoded in-register format that is produced on load from memory. The third is a pipeline format that is consumed by high-performance functions in the pipeline and is decoded on read from the register file.

The open-source cheri-cap-lib library (summarised in Appendix E of [Wat20b]) used in our open-source CHERI implementations expresses these three levels of compression using a typeclass; a microarchitectural "API" to capabilities which includes capability manipulation operations for each level of decompression. The in-register view extracts E (the exponent) into a dedicated field, reconstitutes the top two bits of the T field, and also extracts a_{mid} from the address into a dedicated field. This decompression is fast enough to be performed parallel to the byte-select in the general-purpose loads in the CHERI-MIPS pipeline [Woo19]. In turn, this format is further decoded into the in-pipeline format by adding a few Booleans and 2-bit offsets that locate the top and base with respect to the address. These fields are used to perform capability operations such as computing a full top or base, or fast representability checks.

While many variations of the capability typeclass could be useful for a CHERI implementation, these three design points provided in the library have enabled sufficient flexibility to implement an array of optimised hardware microarchitectures.

4.1.1.3 Program Counter Capability (PCC)

CHERI extends the program counter (PC) with bounds and permissions, which are together called PCC. PC is very performance sensitive and is predicted in most microarchitectures.

A processor requires the address of PCC at the earliest stage of the pipeline to initiate instruction fetch, but the bounds and permissions of PCC are needed only to decide exception conditions and can therefore be checked at any point in the pipeline. The CHERI-MIPS microarchitecture takes advantage of this distinction to speculate on only the address of PCC (i.e., the address of the instruction to be fetched) using the branch predictor, but forwards updates to PCC bounds to the execute stage of the pipeline. This microarchitecture is described in [Woo14]. The in-order Piccolo and Flute microarchitectures share this design.

Forwarding in superscalar and out-of-order pipelines is more complex so we chose to predict the entirety of PCC in the Toooba microarchitecture. The BTB delivers both bounds and address, allowing the bounds to be checked early in the pipeline, with a branch misprediction resulting from a mismatch in either the address bits or the bounds and permissions. The Arm implementers of the Morello prototype observe that predicting the bounds, as is done in Toooba, is the optimal performance solution, though it is expensive microarchitecturally. In order to optimise prediction state, a microarchitecture may choose to predict the bounds separately from the address to take advantage of shared bounds and permissions between branch targets.

Section 4.2 describes Arm's Morello microarchitecture, including PC/PCC prediction, in greater detail. Another way of handling the PC/PCC interlock would be to rename PC/PCC and do physical tag tracking for dependencies, as is done with any other renamed register.

4.1.1.4 Default Data Capability (DDC)

CHERI defines a DDC register which provides both an offset and bounds for non-capability-aware loads and stores such that the capability mechanism can constrain legacy executables. Both the offset and the bounds for integer addresses require special handling in the microarchitecture.

The CHERI-MIPS architecture added register-offset addressing to the base MIPS instruction set such that a capability-address store has two integer register operands (data and offset) as well as a capability register address operand. This path in the CHERI-MIPS microarchitecture was used to implement standard MIPS loads and stores which also offset through DDC.

The CHERI-RISC-V architecture does not introduce a new addressing mode and therefore does not require an additional register operand for addressing in any common case. Our CHERI-RISC-V microarchitectures implement DDC as a special, non-forwarded capability register. That is, CSpecialRW modifications of DDC traverse the back end of the pipeline alone to avoid consistency issues. DDC is then read directly in any place in the pipeline where it is needed directly from the register without forwarding.

In many implementations it may be desirable to optimise the address offset of DDC, as an extra add on the address-generation path may be problematic.

For memory operations with an immediate offset, microarchitectures without DDC forwarding can add the DDC offset to the immediate operand before the Execute stage so that we preserve two-operand address calculation on the critical path. As RISC-V and MIPS exclusively provide immediate-offset addressing, this optimization resolves the issue for these architectures where DDC forwarding is not implemented.

For register-offset addressing or for microarchitectures that forward DDC, the DDC-offset performance might be optimised by an architectural change that limits allowed alignments and possibly sizes of DDC. For example, if DDC were constrained to be aligned and sized to the same power of two, we may simply OR the DDC offset with the resulting address. If the access is in-bounds, the OR will be exactly equivalent to an ADD, and if not the instruction will be cancelled due to the exception. One could also imagine implementations that perform this optimization microarchitecturally such that power-of-two aligned-and-sized DDCs operate at full speed and other values of DDC fall back to a lower performance mode.

4.1.1.5 Capability mode bit

A CHERI ISA may choose to implement a capability *mode* bit in order to reuse legacy load and store encodings for capability-based loads and stores. This mode bit affects the decoding of instructions, determining the source of bounds for memory access and may also determine alternate decodings of other repurposed instructions.

CHERI-RISC-V specifies a mode bit that is defined in PCC. Our in-order open-source implementations, Piccolo and Flute, forward the mode bit along with the bounds to the Execute stage such that mode-bit-dependant decoding does not take place before Execute, which surprisingly fits well into these microarchitectures.

Our Toooba implementation predicts the mode bit with the entirety of PCC so that the mode bit is available anywhere in the pipeline including Decode.

If the mode bit is specified in a special register rather than being attached to PCC, the same design options should be available. The mode bit might cause a pipeline flush on modification so that the current mode might be accessed anywhere in the pipeline, might be forwarded to allow rapid mode switching, or might be predicted along with PCC to optimise repeated mode switches.

4.1.1.6 Bounds checking

Many CHERI instructions must check bounds, including all memory-access instructions and many capability-manipulation instructions. The CHERI-MIPS microarchitecture implements all instructions with a single general-purpose bounds-check unit in the pipeline which is never used more than once by any instruction. In addition to this general-purpose bounds-check, there is a bounds-check on PCC.

The general-purpose bounds check generally produces exceptions and does not affect destination register values, so the bounds-check in CHERI-MIPS is set up in the Execute stage but is performed during Memory Access.

The general-purpose bounds-check unit must support not only less-than comparison for the top (the common case), but also less-than-or-equal-to for CSetBounds. CSetBounds also requires extended precision for the upper bound rather than wrap-around arithmetic used by addresses. The bounds check unit supports separate upper and lower address comparisons against the upper and lower bounds respectively in order to validate the highest and lowest byte accessed by a memory transaction, and also to support the TestSubset operation. This shared-bounds-check strategy is also used in the Piccolo and Flute microarchitectures, and the Toooba superscalar microarchitecture has one set of bounds-check logic per integer pipe and memory access pipe with some specialization to the two cases.

The PCC bounds check can be optimised in a number of ways.

The CHERI-MIPS and CHERI-RISC-V architectures are designed to remove the need to ever perform a *representable check* on PCC modification. While any legacy branch or jump to an integer register could be considered an add to the address of PCC, potentially requiring a representability check, the MIPS and RISC-V CHERI architectures avoid this condition by throwing an out-of-bounds exception on the branch using the general-purpose bounds-check for control flow instructions.

All of our open-source CHERI microarchitectures bounds check PCC for every instruction executed. As branch targets are guaranteed to be in-bounds, it should be possible for the bounds check on PCC to be elided for the majority of instructions, possibly checking PCC bounds in batches. Alternatively, we might calculate the number of instructions to the bound on each jump and assert that the distance-to-the-bound counter does not reach zero.

4.1.1.7 Special Capability Registers (SCRs)

The set of *Special Capability Registers* (SCRs) contain registers with special pipeline implications (such as DDC and PCC) and also registers to allow simplified privilege escalation which are gated by privilege ring. These registers are accessible only through explicit move-from and move-to GPR instructions.

The CHERI-MIPS implementation implements all SCRs besides PCC, KCC, and EPCC as forwarded general-purpose capability registers. PCC is predicted, and KCC and EPCC are used for swapping with PCC on exception and are broken out into dedicated registers without forwarding to enable single-cycle exceptions without the need to access the forwarded register file.

Piccolo, Flute, and Toooba implement all SCRs as non-forwarded registers, blocking the pipeline for the duration of the execution of any SCR modification.

4.1.2 Compressed capability optimizations

The compression scheme used in CHERI-128 and CHERI-64 is partially described in [Woo19] and its key algorithms are implemented in the *cheri-cap-lib* repository [cheri-cap-lib]. The key algorithms are reproduced in Appendix E of [Wat20b]. These algorithms are a significant contribution to the community as they enable reasonably efficient microarchitectural implementations of CHERI.

4.1.2.1 Decompressing bounds

Decompressing the bounds of capabilities is highly optimised. Bounds decompression requires detecting the relative positioning between the top and bottom with respect to the address, as described in [Woo19].

The *GetTop* function is more complex than the *GetBase* function, as described in Section E.1 of [Wat20], due to the requirement to discern between top being 0 or 2^{64}. This algorithm is a noted contribution as it is non-trivial to develop a correct algorithm that is fast enough for common use in pipelines.

4.1.2.2 Bounds checking

There are two types of bounds check in CHERI microarchitectures: precise and representable. Precise bounds checks assert that an address is between the bounds of the capability, but a representable check asserts that a transformation on the address of a compressed capability does not change its bounds due to the limitations of compression.

4.1.2.2.1 Precise bounds check

Cheri-cap-lib provides *CapInBounds* (listed in Section E.2 of [Wat20b]) which is an algorithm to check that an encoded capability is within bounds without decompressing the bounds of the capability.

We have not required an optimised bounds check which integrates an offset to the address (e.g., for offset addressing) as these bounds checks are not on the critical path in our designs, but generally produce exceptions. Precise bounds checks with an offset are usually checked against fully decoded bounds (usually in the next pipeline stage), but could use the *IncOffset* function discussed below followed by *CapInBounds*.

4.1.2.2.2 Fast representability checks

When the address of a capability is being modified, the algorithm must assert that the resulting capability will still decode the same bounds as the original capability. Custom *fast representability checks* that operate directly on the compressed fields of the encoding are required for each single-cycle operation that modifies the address to avoid dependence on decompressed values which generally require the majority of a cycle to calculate.

The *IncOffset* and *SetOffset* operations are supported by a single function listed in Section E.3 of [Wat20b], and implement the function described in Section 3.5.4 of [Wat20b]. This shared function allows a single circuit to support both operations.

The *SetAddress* operation must detect if an arbitrary new address is within the representable limits of the capability, and has a distinct implementation listed in Section E.4 of [Wat20b].

4.1.2.3 Setting bounds

The *SetBounds* function listed in Section E.5 of [Wat20b] and described in Section 3.5.4 of [Wat20b] provides a single, shared, high-speed function that returns a single data structure containing a capability with the new bounds (to implement CSetBounds), a flag indicating if rounding was necessary (to facilitate CSetBoundsExact), a mask that could be applied to a pointer to align it with the supplied length (to facilitate CRepresentableAlignmentMask), as well as the length that was actually achieved after rounding (to facilitate CRoundRepresentableLength).

It is a challenge to implement a *SetBounds* circuit that rounds only when precisely necessary while achieving single-cycle execution. The example algorithm is described in detail in the comments of the listing, and includes pre-computing all fields for both the rounded and unrounded cases while simultaneously detecting if rounding will occur, followed by a select of the correct return values. The rounding detection logic is sophisticated and uses a *smear-right* technique to generate a mask to select bits in the address and length relative to the most significant set bit of the length without waiting for the result of *CountLeadingZeros*. A case matrix is then constructed to detect carry ins to an arbitrary region of the new top based on masked values rather than waiting for the result of full adds.

4.1.3 Loading and storing capabilities

CHERI requires atomic memory access to capability-wide words (e.g., 129-bit words).

4.1.3.1 Capability width

All open-source CHERI implementations have widened the memory interface between the core and the caches to the capability width. The CHERI-MIPS microarchitecture required that all memory paths were at least capability-width at least to the TagController. For Flute and Piccolo we allow memory paths past the L1 to be half the width of a capability, but never split a capability between bursts. These implementations duplicate the tag bit on the two flits of the capability. The *tag* bit is added to the USER fields of the data channels of AXI (RDATA/WDATA) in these implementations, so it was not possible to transfer less than one tag bit with each flit.

In addition to the cache interfaces, the Toooba microarchitecture enlarged all load and store buffers to at least 129 bits along with all other memory forwarding paths.

4.1.3.2 Capability permission complexity

The CHERI architecture requires data-dependent faults such that the address and data must be available for inspection before a store can be issued. Specifically, the architecture defines a PERMIT_STORE_LOCAL_CAPABILITY bit on an address that may trigger a fault if the GLOBAL bit is not set on capability data that is being stored.

The open-source CHERI implementations are based on microarchitectures that do not issue stores unless both operands are available, and so are able to trivially inspect both operands and mark a store for exception if necessary. Most high-performance processors will separate address and data issue to release the store address as soon as possible, but might need to delay the store address issue until store data is available in order to capture both the tag and global bit for fault detection. A few other options could be considered if the PERMIT_STORE_LOCAL_CAPABILITY is no longer required in the architecture.

4.1.4 Tagged memory

The CHERI capability model requires one extra bit per capability word, e.g., CHERI-128 requires 129-bit memory words. This requires changes to the microarchitecture of the memory subsystem to widen structures where possible, or emulate wider memory where it is not.

4.1.4.1 Tagging data caches

Data banks and interfaces of caches can simply be widened to accommodate tagged words. The capability tags can either be stored in the data banks, or the capability tags for a line might be aggregated and stored separately, e.g., into the record for that line in the cache-tag bank. We have used both approaches in open-source implementations, with the second facilitating CLoadTags.

Alternatively, tags could be stored in a separate cache structure that could reduce on-chip storage using compression. This design point would need to solve problems with coherency and would need to integrate into the pipeline such that tag and data pairs are always accessed atomically.

Efficient Tagged Memory [Joa17] discusses each of the above design points in detail.

4.1.4.2 Tagging memory

External memory has become a commodity, so there are strong pressures to build systems that support industry-standard interfaces.

4.1.4.2.1 Tag controller with cache

Our primary approach, described in [Joa17], is a tag controller that allows an external memory controller to emulate a memory holding tagged words. This tag controller maintains a tag table in the external memory and provides the tag bits for each line requested from the remainder of external memory. The tag controller contains a cache of lines from the tag table to reduce tag table accesses. Furthermore, the tag table can be hierarchical such that each bit of a root level indicates whether any bits are set in a block of the leaf table. This structure significantly reduces cache pressure, as a single line of the root table can potentially replace many lines of the leaf (or flat) table.

We have implemented this approach in the open-source TagController project which is used in all of our open-source implementations. This tag controller is parametrisable for arbitrary hierarchy depths and arbitrary block sizes at each level of the hierarchy.

4.1.4.2.2 Wide memory

Commodity external memory may hold memory words wider than its processor word size (e.g., ECC memory). A capability system may choose such a memory type and use these bits to hold capability tags alongside data in external memory. As ECC bits typically provide more storage than necessary to hold capability tags, a CHERI system using ECC memory should be able to support in-word capability tags and as well as error detection and correction at some level.

4.1.4.2.3 Dedicated memory

It is also possible to design a system with a dedicated memory channel for tags. For example, a system with a 1024-bit memory interface (e.g., HBM) might add an 8-bit memory interface for accessing tags.

4.1.4.3 Loading tags

The CHERI architecture includes a CLoadTags instruction to load the tags for a cache line into a register. CLoadTags is expected to be cache coherent, so it is not possible to bypass data caches completely, and it is complex to allow greater-than-cache-line granularity

The CHERI-MIPS implementation will opportunistically return tags from the cache if the line is present, but will not trigger a cache fill based on a miss due to CLoadTags, but will forward the request to the next level of cache hierarchy, ultimately hitting the TagController if all caches miss. If the request hits the TagController, it will respond to the request directly, performing an ordinary tag table lookup and caching any results.

4.1.5 Speculative side-channel precautions

We recommend that CHERI implementations take reasonable precautions to prevent data access through speculative side channels. Many of these observations are explored in [Wat18]. CHERI microarchitectures should reasonably be expected to constrain memory access in speculation to capabilities that are architecturally available to the program. That is, no capability should exist in registers and forwarding paths beyond capabilities in the latest committed architectural state of the register file, those transitively reachable through them, and less-powerful capabilities derived from these. This property, which we may call Speculative Capability Constraint (*SCC*), requires a few microarchitectural features.

4.1.5.1 Capability-guarded cache access

Permissions and bounds of a memory access should be verified before any cache access is initiated. This is generally reasonable, as these checks are simpler than TLB translation which must also occur before the cache can take action on behalf of a memory access. This property is necessary to support *SCC* by preventing unreachable capabilities from being introduced into the pipeline from memory. All of our open-source implementations have this behaviour.

4.1.5.2 PCC bounds forwarding (not prediction)

SCC may be violated by a design that predicts the bounds of PCC. For example, the Toooba implementation predicts the bounds of PCC, storing the entire PCC in the branch target buffer. On any jump, it is possible for a powerful PCC to be predicted, introducing read rights to new addresses that were not implied by the latest-committed state of the register file. The Morello implementation chooses to forward the bounds of PCC rather than predict them, so the PCC capability cannot be used in a data memory access unless it is legally sourced form another register in the pipeline. CHERI-MIPS, Piccolo, and Flute share this design choice, though they are of less note as their simple pipelines do not allow speculative read gadgets.

4.1.5.3 Speculative forgery prevention

SCC may also be violated if capabilities can be forged in speculation. CHERI capability manipulation instructions should not speculatively produce capabilities with privilege greater than their operands provide. For example, CBuildCap should not forward tagged bits to its consumers while waiting for the result of its bounds check. CSetBounds and CUnseal share similar concerns. It is believed that all open-source CHERI implementations may currently forward unsafe values for these instructions, and the Toooba microarchitecture is likely vulnerable to speculative execution attacks through this vector. This concern might be more systematically alleviated by an architecture that clears tags rather than throws exceptions for operations that manipulate the privilege of a capability. The Arm Morello architecture generally clears tags rather than throws exceptions for capability manipulation instructions, and we expect the Morello microarchitecture to be immune to this class of attacks for this reason.

In addition, any speculation in a microarchitecture that could synthesise a value rather than deriving it from architecturally defined bits (e.g., value speculation) should not produce valid capability values. This could include not only a capability predicted to be loaded from a memory location, but also a predicted integer value that is used to bound a capability.

If these restrictions become performance-limiting, one could imagine deploying Speculative Taint Tracking[1] ensure that speculatively forged capabilities do not affect cache state.

4.1.5.4 Compartment ID (CID) enforcement

Even if *SCC* is enforced, we may have code paths that manipulate powerful capabilities that must be protected from speculative execution attacks. The CID is described in Section 2.5 of [Wat20b] to provide an architectural means to convey trust boundaries to the microarchitecture. The CID should be used to tag microarchitectural state to prevent instructions in disparate compartments influencing each other's execution in much the same way as we might hope a modern microarchitecture might prevent user space speculative state influencing kernel execution. For example, the branch target buffer should tag entries with the CID such that targets learned in one compartment would not be used when speculating in another compartment, allowing an attacker to redirect branches that expose powerful capabilities to side-channel gadgets. The branch history table may also be tagged, as well as prefetchers and any other structure that holds state that influences prediction.

The CID itself may be large and require compression in the microarchitecture. For example, a microarchitecture may introduce a table that holds several active CIDs while attaching table indices to all state used for speculation. Such a microarchitecture requires a means to flush state belonging to old table values before installing a new value at an index.

If domain crossing is to be highly optimised, the application of the CID may be imprecise (e.g., allowing use of old CID state until the new CID install commits) or the CID itself may be predicted. Either of these may be very difficult to allow without introducing speculative-side- channel vulnerabilities.

4.2 Morello

In addition to microarchitectural IP listed above in CHERI based implementations by University of Cambridge, following microarchitectural ideas in the Arm's Morello implementation could be considered essential IP.

4.2.1 ST(L)XP

Wider memory operations due to capability-width transactions may require cracking instructions into multiple μops (micro-operations), which in turn may require atomically tying together multiple memory buffer entries. For example, ST{L}XP is a 32B operation. Morello implemented this in a 16B store buffer design which required cracking the operation into two μops. This required complexity to handle correct success/failure for two μops instead of one to make sure the result is atomic (both succeed or both fail). Logic had to handle one tag/address for both stores, had to handle translation atomicity, and had to handle writing data from both or none at all. Morello used adjacent pairs to associate the merge buffer entries together so that they always retire together and write together or fail together.

Another microarchitecture implementation could use a simplified design where the entire store data path is 32B within the processor – store buffers, writes to the cache or memory subsystem.

---

[1] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas and C. W. Fletcher. 'Speculative Taint Tracking (STT) A Comprehensive Protection for Speculatively Accessed Data'. In: Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (IEEE MICRO 2019) (Columbus, Ohio, USA). Oct. 2019.

Other solutions are very dependent on the microarchitecture implementation of stores: how and where they track pass/fail of the exclusive instructions; how they write to the data cache/memory subsystem; how they keep ordering of stores (for release semantics).

4.2.2 LDCT – read all tag bits for a cache line:

If the tag bits are stored in the same banks as the data they are associated with, and on a cache miss, the data returned is sent in multiple beats, then special care has to be made to wait for all the data to return. For example, in Morello, each cache line is filled in two beats, and we were able to ensure that the entire line was filled before responding by setting the access address to the centre of the cache line including bytes from both beats, forcing the load operation to wait until the entire cache line was returned.

Other implementations:

For ideal LDCT performance, you would want the tags to be tracked separately and not require any special handling.

With our implementation memory model, another way to implement LDCT is to crack it into 4 tag loads and merge their results.

4.2.3 PC/PCC dependency tracking

In legacy designs, the PC is predicted and therefore assumed to be known from the beginning of the pipeline for every instruction. In contrast, the Bounds/etc of the PCC do not need to be known up front, and do not need to be predicted, which would be expensive and/or complex.

Rather than predicting the bounds/etc of PCC, Morello implements a custom forwarding mechanism for operations that read their PCC, including both ALU operations and commit-time PC bounds checking. Morello keeps the bounds of in-flight PCCs in a dedicated register file and each instruction is associated with an entry. At commit time, the bounds of each instruction must be known to resolve any PCC bounds exceptions. In addition, any instructions that take their PCC as an operand will block in the issue queue until their PCC is resolved.

In fact, the Morello base architecture kept PC base addresses in a centralised PC regfile (PCRF), and any PC readers require an index into this register file to resolve the upper bits of the PC. In Morello, these PCRF entries held a further index into a bounds register file, PCCRF which is only populated as branches are actually resolved. For commit time bounds checking, we walk and broadcast the PCRF entries that have a “resolved” (known) PCCrf entry so we can resolve any PCC bounds exceptions.

The issue queues can use this broadcast PCRF to release the dependency on the PCC that any PCC readers have without having to rename the PC or track any extra information.

Other implementations:

Predicting the bounds would be the optimal performance solution (though expensive).

Another way of handling the PC/PCC interlock would be to rename the PC/PCC and do physical tag tracking for dependencies (like any other renamed register).

5 Bibliography and related work

5.1 Peer-reviewed articles and papers

[Arm19] 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. **ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS**. In POPL 2019, Proc. ACM Program. Lang. 3, POPL, Article 71.

[Chi15] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. **Beyond the PDP-11: Architectural support for a memory-safe C abstract machine**, Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015), Istanbul, Turkey, March 2015.

[Chi17] David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A. Theodore Markettos, J. Edward Maste, Robert Norton, Stacey Son, Michael Roe, Simon W. Moore, Peter G. Neumann, Ben Laurie, and Robert N. M. Watson. **CHERI-JNI: Sinking the Java security model into the C**. Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2017). Xi'an, China, April 8–12, 2017.

[Dav19a] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. **CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment**. In Proceedings of 2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS'19). Providence, RI, USA, April 13-17, 2019

[Fil20] 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. **Cornucopia: Temporal Safety for CHERI Heaps**. In Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland 2020). San Jose, CA, USA, May 18-20, 2020.

[Joa17] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W. Moore, Alex Bradbury, Hongyan Xia, Robert N. M. Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alex Richardson, Stacey Son, and A. Theodore Markettos. **Efficient Tagged Memory**. Proceedings of the 2017 IEEE 35th International Conference on Computer Design (ICCD). Boston, MA, USA, November 5-8, 2017.

[Maz18] Alfredo Mazzinghi, Ripduman Sohan, and Robert N. M. Watson. **Pointer Provenance in a Capability Architecture**. Proceedings of the 10th USENIX Workshop on the Theory and Practice of Provenance (TaPP'18). London, UK, July 11-12, 2018.

[Mem16] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. **Into the depths of C: elaborating the de facto standards**. Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016). Santa Barbara, CA, USA, June 2016.

[Mem19] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. **Exploring C Semantics and Pointer Provenance**. In Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), Cascais, Portugal, 13-19 January, 2019.

[Neu10] Peter G. Neumann, Robert N.M. Watson. **Capabilities Revisited: A Holistic Approach to Bottom-to-Top Assurance of Trustworthy Systems**. Proceedings of the Fourth Annual Layered Assurance Workshop, Austin, Texas, December 2010.

[Neu17] Peter G. Neumann. **Fundamental Trustworthiness Principles**, in New Solutions for Cybersecurity, Howie Shrobe, David Shrier, Alex Pentland, eds., MIT Press/Connection Science: Cambridge MA.

[Nie20] Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. **Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process**. In Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland 2020). San Jose, CA, USA, May 18-20, 2020.

[Wat12] Robert N.M. Watson, Peter G. Neumann Jonathan Woodruff, Jonathan Anderson, Ross Anderson, Nirav Dave, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Philip Paeps, Michael Roe, and Hassen Saidi. **CHERI: a research platform deconflating hardware virtualization and protection**. Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE 2012), March, 2012.

[Woo13] Jonathan Woodruff, Simon W. Moore and Robert N.M. Watson, **Memory Segmentation to Support Secure Applications**, CEUR Workshop: Doctoral Symposium on Engineering Secure Software and Systems (ESSoS), Paris, France, 26–27 February, 2013.

[Wat15] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. **CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization**, Proceedings of the 36th IEEE Symposium on Security and Privacy ("Oakland"), San Jose, California, USA, May 2015.

[Wat16] Robert N.M. Watson, Robert M. Norton, Jonathan Woodruff, Simon W. Moore, Peter G. Neumann, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Michael Roe, Nirav H. Dave, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Ed Maste, Steven J. Murdoch, Colin Rothwell, Stacey D. Son, and Munraj Vadera. **Fast Protection-Domain Crossing in the CHERI Capability-System Architecture**. IEEE Micro vol. 36 no. 5, p. 38-49, Sept.-Oct., 2016

[Wat17] Robert N. M. Watson, Peter G. Neumann, and Simon W. Moore. **Balancing Disruption and Deployability in the CHERI Instruction-Set Architecture (ISA)**, New Solutions

for Cybersecurity, Shrobe H., Shrier D., Pentland A. eds., MIT Press/Connection Science: Cambridge MA.

[Woo14a] 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. **The CHERI capability model: Revisiting RISC in an age of risk**, Proceedings of the 41st International Symposium on Computer Architecture (ISCA 2014), June 14–16, 2014, Minneapolis, MN, USA.

[Woo19] 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. **CHERI Concentrate: Practical Compressed Capabilities**. In IEEE Transactions on Computers, 10.1109/TC.2019.2914037, IEEE, 2019.

[Xia18] Hongyan Xia, Jonathan Woodruff, Hadrien Barral, Lawrence Esswood, Alexandre Joannou, Robert Kovacsics, David Chisnall, MichaelRoe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alex Richardson, Simon W. Moore, and Robert N. M. Watson. **CheriRTOS: A Capability Model for Embedded Devices**. Proceedings of the 2018 IEEE 36th International Conference on Computer Design (ICCD). Orlando, FL, USA, October 7-10, 2018.

[Xia19] 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. **CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety**. In Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (IEEE MICRO 2019). Columbus, Ohio, USA, October 12-16, 2019.

5.2 Technical reports and specifications

[Arm20] Arm Limited. **Arm® Architecture Reference Manual Supplement Morello for A-profile Architecture**. Document DDI060, Version A.g. 2020.

[Dav19b] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. **CheriABI: Enforcing valid pointer provenance and minimizing pointer privilege in the POSIX C run-time environment**, Technical Report UCAM-CL-TR-932, Computer Laboratory, January 2019.

[Nie19] Kyndylan Nienhuis, Alexandre Joannou, Anthony Fox, Michael Roe, Thomas Bauereiss, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. **Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process**. Technical Report UCAM-CL-TR-940, University of Cambridge, Computer Laboratory, September 2019.

[Wat18] Robert N. M. Watson, Jonathan Woodruff, Michael Roe, Simon W. Moore, Peter G. Neumann. **Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the Meltdown and Spectre Attacks**, Technical Report UCAM-CL-TR-916, Computer Laboratory, February 2018.

[Wat19a] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Nathaniel Wesley Filardo, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Rugg, Peter Sewell, Stacey Son, Hongyan Xia. **Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7)**, Technical Report UCAM-CL-TR-927, Computer Laboratory, June 2019. Current CHERI ISA specification

[Wat19b] Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Peter G. Neumann. **An Introduction to CHERI**, Technical Report UCAM-CL-TR-941, Computer Laboratory, September 2019.

[Wat20a] Robert N. M. Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W. Moore, Edward Napierala, Peter Sewell, and Peter G. Neumann. **CHERI C/C++ Programming Guide**, Technical Report UCAM-CL-TR-947, Computer Laboratory, June 2020.

[Wat20b] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, Hongyan Xia. **Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)**, Technical Report UCAM-CL-TR-951, Computer Laboratory, October 2020.

5.3 PhD dissertations

[Joa19] Alexandre J. P. Joannou. **High-performance memory safety: optimizing the CHERI capability machine**, Technical Report UCAM-CL-TR-936, University of Cambridge, Computer Laboratory, May 2019.

[Nor16] Robert M. Norton. **Hardware support for compartmentalisation**, Technical Report UCAM-CL-TR-887, University of Cambridge, Computer Laboratory, May 2016.

[Ric20] Alexander Richardson. **Complete spatial safety for C and C++ using CHERI capabilities**, Technical Report UCAM-CL-TR-949, Computer Laboratory, June 2020.

[Wat12] Robert N. M. Watson. **New approaches to operating system security extensibility**, Technical Report UCAM-CL-TR-818, University of Cambridge, Computer Laboratory, April 2012.

[Woo14b] Jonathan D. Woodruff. **A RISC capability machine for practical memory safety**, Technical Report UCAM-CL-TR-858, University of Cambridge, Computer Laboratory, July 2014.

5.4 Open-source models, properties, and proofs

[sail-cheri-riscv] CHERI Project. **CTSRD-CHERI/sail-cheri-riscv: Sail formal model of the CHERI-RISC-V ISA**. URL: https://github.com/CTSRD-CHERI/sail-cheri-riscv.

[sail-cheri-mips] CHERI Project. **CTSRD-CHERI/sail-cheri-riscv: Sail formal model of the CHERI-MIPS ISA**. URL: https://github.com/CTSRD-CHERI/sail-cheri-mips.

5.5 Open-source hardware implementations

[cheri-cap-lib] CHERI Project. **CTSRD-CHERI/cheri-cap-lib: A library of specific implementations of cheri and providing an abstract interface to those implementations**. URL: https://github.com/CTSRD-CHERI/cheri-cap-lib.

[CHERI-MIPS] CHERI Project. **CTSRD-CHERI/cheri-cpu: CHERI-MIPS implementation in a 6-stage pipeline with asso- ciative caches and multi-core support**. URL: https://github.com/CTSRD-CHERI/cheri-cpu.

[Flute] CHERI Project. **CTSRD-CHERI/Flute: RISC-V CPU, simple 5-stage in-order pipeline, for low-end applic- ations needing MMUs and some performance**. URL: https://github.com/CTSRD-CHERI/Flute.

[Piccolo] CHERI Project. **CTSRD-CHERI/Piccolo: RISC-V CPU, simple 3-stage pipeline, for low-end applications (e.g., embedded, IoT)**. URL: https://github.com/CTSRD-CHERI/Piccolo.

[TagController] CHERI Project. **CTSRD-CHERI/TagController: Multi-level tag controller for emulating a tagged memory using an in-memory table**. URL: https://github.com/CTSRD-CHERI/TagController.

[Toooba] CHERI Project. **CTSRD-CHERI/Toooba: RISC-V Core; superscalar, out-of-order, multi-core capable; based on RISCY-OOO from MIT**. URL: https://github.com/CTSRD-CHERI/Toooba.

5.6 Open-source software implementations

[CheriBSD] CHERI Project. **CTSRD-CHERI/cheribsd: CHERI-adapted version of the FreeBSD operating system**. URL: https://github.com/CTSRD-CHERI/cheribsd.

[CHERI Clang/LLVM] CHERI Project. **CTSRD-CHERI/llvm-project: CHERI adaptation of the Clang/LLVM compiler and LLD linker**. URL: https://github.com/CTSRD-CHERI/llvm-project.

[CHERI GDB] CHERI Project. **CTSRD-CHERI/gdb: CHERI adaptation of the GDB debugger**. URL: https://github.com/CTSRD-CHERI/gdb.

[QEMU-CHERI] CHERI Project. **CTSRD-CHERI/qemu: CHERI adaptation of the QEMU ISA emulator**. URL: https://github.com/CTSRD-CHERI/qemu.