Verified Security for the Morello Capability-enhanced Prototype Arm Architecture

Verified Security for the Morello Capability-enhanced Prototype Arm Architecture

Thomas Bauereiss[1]✉, Brian Campbell[2], Thomas Sewell[1], Alasdair Armstrong[1], Lawrence Esswood[1], Ian Stark[2], Graeme Barnes[3], Robert N. M. Watson[1], and Peter Sewell[1]

[1] University of Cambridge, Cambridge, UK
first.last@cl.cam.ac.uk
[2] University of Edinburgh, Edinburgh, UK
first.last@ed.ac.uk
[3] Arm Ltd., Cambridge, UK
first.last@arm.com

Abstract. Memory safety bugs continue to be a major source of security vulnerabilities in our critical infrastructure. The CHERI project has proposed extending conventional architectures with hardware-supported *capabilities* to enable fine-grained memory protection and scalable compartmentalisation, allowing historically memory-unsafe C and C++ to be adapted to deterministically mitigate large classes of vulnerabilities, while requiring only minor changes to existing system software sources. Arm is currently designing and building Morello, a CHERI-enabled prototype architecture, processor, SoC, and board, extending the high-performance Neoverse N1, to enable industrial evaluation of CHERI and pave the way for potential mass-market adoption. However, for such a major new security-oriented architecture feature, it is important to establish high confidence that it does provide the intended protections, and that cannot be done with conventional engineering techniques.

In this paper we put the Morello architecture on a solid mathematical footing from the outset. We define the fundamental security property that Morello aims to provide, reachable capability monotonicity, and prove that the architecture definition satisfies it. This proof is mechanised in Isabelle/HOL, and applies to a translation of the official Arm specification of the Morello instruction-set architecture (ISA) into Isabelle. The main challenge is handling the complexity and scale of a production architecture: 62,000 lines of specification, translated to 210,000 lines of Isabelle. We do so by factoring the proof via a narrow abstraction capturing essential properties of arbitrary CHERI ISAs, expressed above a monadic intra-instruction semantics. We also develop a model-based test generator, which generates instruction-sequence tests that give good specification coverage, used in early testing of the Morello implementation and in Morello QEMU development, and we use Arm's internal test suite to validate our model.

This gives us machine-checked mathematical proofs of whole-ISA security properties of a full-scale industry architecture, at design-time. To the best of our knowledge, this is the first demonstration that that is feasible, and it significantly increases confidence in Morello.

1 Introduction

Memory safety bugs continue to be a major source of security vulnerabilities, responsible for around 70% of those addressed by Microsoft security updates, and around 70% of the high-severity bugs impacting Chromium [30,14]. Their root causes are well-known legacy design choices and limitations of normal practice: pervasive uses of systems programming languages that do not enforce memory protection; hardware that enforces only coarse-grain protection, using virtual memory; and test-and-debug development methods that cannot provide high assurance. These are baked in to the critical systems codebase across the industry, and the result, in today's adversarial environment, is that programming errors can often lead to exploitable vulnerabilities.

There are many possible approaches to improving this situation, including development of safer programming languages, techniques for full functional-correctness verification, and better bug-finding tools. Each is the subject of much research in programming languages and semantics, and all are worthwhile, but the legacy investment, the need for systems code to work close to the machine, and the inability of bug-finding to provide high assurance, have made it very hard to radically improve mass-market systems.

Another path, less well explored, is to change the architectural interface to provide hardware mechanisms that enable better enforcement of memory protection. Over the last twelve years, the CHERI project [1] has been extending conventional hardware Instruction-Set Architectures (ISAs) with new architectural features to enable fine-grained memory protection and highly scalable software compartmentalisation. The CHERI memory protection features allow historically memory-unsafe programming languages such as C and C++ to be adapted to have quite different semantics, replacing many unpredictable undefined behaviour (UB) cases with predictable fail-stop traps, to provide strong and efficient protection against many currently widely exploited vulnerabilities. Crucially, this requires only minor changes to the sources of existing systems software. The CHERI scalable compartmentalisation features enable the fine-grained decomposition of operating-system (OS) and application code, to limit the effects of security vulnerabilities.

CHERI provides these via hardware support for *unforgeable capabilities*: in a CHERI ISA [54], instead of using simple 64-bit machine-word virtual-address pointer values to access memory, restricted only by the memory management unit (MMU), one can use 128+1-bit capabilities that encode a virtual address together with the base and bounds of the memory it can access. Encoding these within the capability enables a fast access-time check, faulting if there is a safety violation. A one-bit tag per capability-sized and aligned unit of memory, cleared in the hardware by any non-capability write and not directly addressable, ensures capability integrity by preventing forging, and the ISA design lets code shrink capabilities but never grow them. This architectural mechanism, along with additional sealed-capability features for secure encapsulation, can be used by programming language implementations and systems software in many ways.

Previous academic work on CHERI has developed CHERI-MIPS and CHERI-RISC-V architectures, FPGA processor implementations, and system software including adaptions of Clang/LLVM, linkers, debuggers, FreeRTOS, FreeBSD, and WebKit. The CHERI processor prototypes implement techniques such as compressed capability bounds [58], and a tag controller and cache [26] required to implement memory tagging on off-the-shelf DRAM. The software prototypes use CHERI's architectural features to implement memory-safe CHERI C/C++ programming languages [55], fine-grained spatial memory safety [15], heap temporal memory safety [15], and scalable software compartmentalisation [57]. An analysis of vulnerabilities reported to the Microsoft Security Response Center (MSRC) in 2019 suggested that CHERI memory safety would have deterministically mitigated 30%–70%, depending on the usage scenario [27], and porting the FreeBSD kernel and userspace to CHERI required changes only to 0.18% and 0.04% LoC respectively. Analysis of an open-source desktop stack [53] estimated a 73.8% vulnerability mitigation rate through a combination of memory protection and software compartmentalisation requiring a 0.026% LoC change.

Achieving widespread adoption of any substantial new architectural feature is also challenging, of course, but the issues differ from those for adoption of a new high-level programming language. It needs coordinated hardware and software change, which is hard to arrange, but on the plus side there are very few architecture vendors, so if a feature becomes (say) part of the mainline Arm architecture, and there is pull from major partners, then it will be implemented in all conforming Arm implementations and become ubiquitously available in devices. For CHERI, the academic results are encouraging, but achieving such adoption first needs an industry-scale evaluation of a high-performance silicon processor implementation and software stack above it, to demonstrate viability and enable that pull. This is beyond what can be done academically, but hard to justify as a purely commercial project. The 2019–24 UKRI Digital Security by Design (DSbD) challenge resolves this chicken-and-egg difficulty with a combined public-sector and industry (£70m+117m) programme to build and evaluate such demonstration platform, and support research and development above it [52].

Arm, supported in part by DSbD, is currently designing and building Morello, a CHERI-enabled prototype architecture, processor, system-on-chip (SoC), and development board, extending the Armv8.2-A architecture and the high-performance Neoverse N1 processor [6,8]. The Morello processor and SoC implement the CHERI ISAv8 protection model, and utilise CHERI's compressed capability bounds and tagged memory approaches. As of 2021-01, the architecture, emulators, initial development boards with Morello silicon, and initial software toolchains, have all been developed. This will allow evaluation of the CHERI mechanisms in a variety of configurations and use cases on a state-of-the-art hardware platform, and paves the way for the potential adoption of CHERI into future production architectures and devices.

In this paper, we describe work to put the Morello architecture and its security properties on a solid mathematical footing from the outset, and to use semantics to ease conventional engineering.

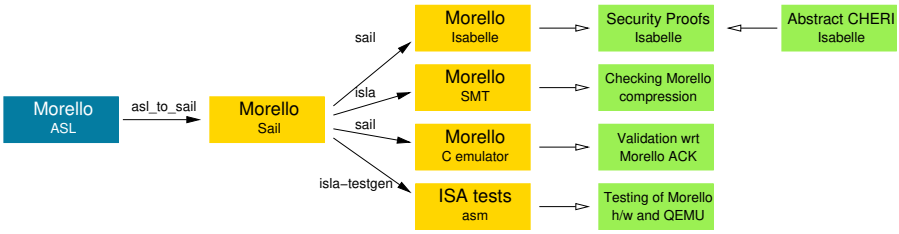


Fig. 1. From Morello ASL source (blue) to auto-generated artifacts (yellow) and verification outcomes (green)

For a new architecture that aims to provide security guarantees, it is especially important to provide high assurance that it actually does. Otherwise, any security flaw in the architecture will be present in any conforming hardware implementation, quite likely impossible to fix or work around after deployment, and the resulting loss of confidence might make further adoption impossible.

For Morello, this is challenging in two ways. First, CHERI needs to be deeply integrated into each base architecture it gets adapted to, most obviously by modifying all virtual-memory-accessing instructions to check bounds and permissions of capabilities, and by adding instructions to explicitly manipulate capabilities, but also in more subtle ways relating to exceptions, virtualisation, and so on. Second, the architecture specification is large and complex. The base Armv8-A architecture is defined in an 8200-page manual [7], to which the Morello architecture supplement adds 1200 more [8]. Fortunately, Arm have recently shifted to using an executable version of their ASL language for instruction-set architecture specification [40,41]. The sequential behaviour is all defined in ASL, and this is what appears in instruction descriptions and auxiliary functions (e.g. for capability compression and address translation) in the documentation. However, it remains very large, 62 000 non-whitespace lines of specification (LoS), and ASL does not itself have a mechanised semantics.

The main intended security property of the Morello architecture is *reachable capability monotonicity*, with the intuition that the available capabilities cannot be increased during normal execution (i.e., they are monotonically decreasing). This is a whole-system property about arbitrary machine execution, and conventional techniques cannot provide high assurance that the architecture satisfies it. Instead, it needs proof. We translate the Arm ASL definition via the Sail [9] language into Isabelle/HOL [39], extending previous work for Armv8-A, and give a mechanised statement and proof that the property holds of the architecture.

We deal with the challenge of scale by factoring the proof via a narrow abstraction: four relatively simple properties of arbitrary CHERI instruction execution that capture essential aspects of their behaviour. Our intra-instruction semantics focusses on the behaviour of instructions in isolation, interacting with registers and memory, rather than viewing each thread as a single state machine; this monadic interface lets us conveniently express these abstract-CHERI properties of instructions in terms of their register and memory effects. We prove

capability monotonicity for arbitrary sequences of instructions above this abstraction, and we instantiate the abstraction for Morello and prove that its many instructions satisfy the required properties. Manual proof effort was required for a number of helper functions defined in the architecture for manipulating and using capabilities, but the bulk of the architecture is handled by automatic proof tools and tactics. Previous work by Nienhuis et al. [38] proved similar results for the much simpler and smaller (6k LoS) CHERI-MIPS architecture with a different approach, manually defining a larger set of abstract actions and proving that those do abstract the instruction semantics. That let one capture instruction intentions more explicitly, but needed more ad hoc machinery, while the new approach we follow here handles the 10x scale-up successfully.

Our proof was developed while the architecture and hardware design were still evolving, using weekly snapshots of Arm's ASL specification, with our automation letting us quickly adapt to changes. This let us identify a number of bugs that could be fixed before the architecture and hardware were finalised.

To validate the ASL-to-Sail translation of the Morello specification, we used the C emulator automatically generated from the Sail model to compare it against Arm's internal Architecture Compliance Kit (ACK) test suite.

Finally, we developed a test generator, using the Isla symbolic execution tooling for Sail [10], to automatically generate interesting instruction-sequence tests, aiming at good specification coverage. These complemented Arm's test suite and were used by Arm as part of their pre-tape-out validation, and were used as the main test suite for development of a Morello version of the QEMU emulator. This helped uncover some bugs in our own tooling as well as discrepancies between different Morello models and emulators. We also used Isla and an earlier Sail-to-SMT flow for quick checking of properties of capability compression.

To summarise, our contributions are:

- A formal and executable semantics of the Morello ISA (§3), automatically translated from the Arm ASL to Sail, Isabelle, and C, and validated against the Arm ACK (§6).
- An abstract characterisation of the essential properties of CHERI ISA instructions, expressed over their intra-instruction semantics (§4).
- A mechanised proof of capability monotonicity for the full sequential Morello ISA specification (including all instructions, system registers, capability compression, etc.), with large parts of the proof automatically generated, making the proof more maintainable as the architecture was developed (§5).
- Automatic ISA test generation from the specification (§7).

This gives us machine-checked mathematical proofs of whole-ISA security properties of a full-scale industry architecture, at design-time. To the best of our knowledge, this is the first demonstration that that is feasible, and it significantly increases confidence in Morello.

The main proof took only around 24 person-months, by two people between 2020-03 and 2021-07, following around 23 person-months of preliminary work to get the model into usable Sail and Isabelle forms, to develop our CHERI abstraction in the context of earlier CHERI architectures, and on our Sail-to-

SMT flow. Test generation and ACK validation took an additional 17 person-months, including Morello-specific work on Isla. This suggests that such proof could be not just technically but also economically viable for new architecture design, particularly as doing this routinely, as an established flow, would reduce the effort substantially.

As a side benefit, our well-validated Morello semantics is reusable for future software or hardware verification. The Armv8-A ISA is, along with x86, one of the two most important low-level programming languages, and if Morello is successful, then one would expect CHERI extensions to be similarly widely used.

Sail and Isabelle versions of the Morello specification, as well as our definitions and proofs, are available online [3].

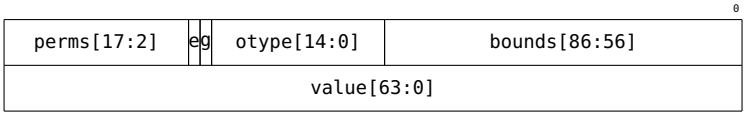
Non-goals and limitations (1) Our results establish confidence that the Morello instruction set architecture design satisfies its fundamental intended security properties. We do not address correctness of the Morello hardware implementation of that architecture, which would be an extremely challenging hardware verification task, and we do not cover system components that are not specified by the ISA itself, e.g. the Generic Interrupt Controller (GIC). (2) The architecture, as usual, expresses only functional correctness properties, not timing or power properties, to allow hardware implementation freedom. Properties and proofs about the architecture therefore cannot address side channels, but see [56] for discussion of side-channels and CHERI. (3) We consider only the sequential architecture. Studying concurrency effects would require a more complex system model integrating the Morello sequential semantics with a whole-system concurrency memory model, which we leave to future work, but we expect the capability properties to be largely orthogonal to concurrency issues, as long as the write of a capability body and tag appear atomic. (4) We assume an arbitrary but fixed translation mapping. CHERI capabilities are in terms of virtual addresses, so system software that manages translations has to be trusted or verified. We also assume that the privileged capability creation instructions are disabled and no external debugger is active, because these features can in general be used to circumvent the capability protections, as discussed in §5.1. (5) Our capability monotonicity property is the most fundamental property one would expect to hold of a CHERI architecture, but it is by no means the only such property. However, stronger properties typically involve specific software idioms, e.g. calling conventions or exception handlers, and their proofs use techniques that have not yet been scaled up to full architectures. We return to this in §8. (6) We prove monotonicity of the Morello specification formally in Isabelle, however, our proof depends on an SMT solver as an oracle for one lemma, as discussed in §5. (7) Our conversion from ASL via Sail to Isabelle is not subject to verification, as neither ASL nor Sail have an independent formal semantics – their semantics is effectively defined by this translation. However, it is nontrivial, and there is the possibility of mismatches with the Sail-generated C emulator used for validation; we do not attempt to verify that correspondence. (8) The ASL specification is subject to the limitations documented by Arm in [7, Appendix K14], e.g. with respect to implementation-defined behaviour.

2 Overview of the Morello CHERI Architecture

CHERI is an architectural protection model that extends ISAs with a new data type, the *architectural capability* [54]. The Morello architecture adds CHERI capabilities to Armv8.2-A, the ISA implemented by the Neoverse N1 CPU on which the Morello hardware implementation is based [8].

2.1 CHERI Capabilities on Morello

CHERI capabilities are twice the natural address size of the architecture plus an out-of-band tag bit, which is not independently addressable; for Morello, capabilities are 128+1 bits. The lower 64 bits are the *"value"*, which in most cases represents a virtual address. The upper 64 bits encode metadata, including bounds, permissions, and other mechanisms. The tag provides integrity protection: it is preserved only by legitimate operations on capabilities, and cleared by others. A capability can only be used as such, e.g. for a dereference, if its tag is set.



A sophisticated compression scheme allows a capability to include 64-bit lower and upper virtual-address bounds, encoded into 87 bits in total, with 56 of those shared with the value field (see [8, §2.5.1],[58] for details). Small regions can be described precisely, with an arbitrary size in bytes, while for larger regions, only certain bounds and sizes are expressible. The capability value must be either within the bounds or within a certain range above or below, allowing for common C idioms that transiently construct (but do not dereference) slightly out-of-bounds pointers; other combinations of value and bounds are not representable. This scheme trades off bounds precision for reduced capability size: supporting arbitrary bounds would require more than 128+1 bits per capability, which would have unacceptable performance costs.

Four of the 18 permission bits are reserved for software, while the others have architecturally defined meaning. The Load, Store, and Execute permissions control whether a capability can be used for loading or storing data or fetching instructions. Permission bits for loading and storing capabilities, as opposed to data, also exist. The System permission controls access to system registers and operations, in addition to the access control mechanisms of the base Arm architecture. Capabilities can also be *sealed*, making them immutable and unusable for anything but branching to them; this allows controlled transitions between different security domains. Sealing (or unsealing) a capability requires an authority capability with the Seal (or Unseal) permission; more on this below.

2.2 Capabilities in Registers and Memory

Morello extends the Armv8-A general-purpose integer register file, as well as certain control and status registers, from 64 bits to 128+1 bits. Memory is extended with a tag bit for each 128-bit sized and aligned unit of DRAM.

The Program Counter (PC) is extended to become a *Program-Counter Capability* (PCC), constraining instruction fetch as well as PC-relative loads (e.g., of global variables). A new *Default Data Capability (DDC)* special register controls and transforms memory accesses relative to machine-word pointer values by legacy (non-capability) instructions, for legacy code using integer pointers.

2.3 Capability-aware Instructions

Morello extends Armv8-A with new instructions and modifies existing instructions to use and respect capabilities. For example, a Load capability (literal) instruction `LDR <Ct>,<label>` calculates an address from the PCC value and an immediate offset, loads a capability from memory, and writes it to capability register `Ct` [8, §4.4.76]. If the PCC capability does not have the load permission, or the calculated address is outside its bounds, a capability fault exception is raised. The tag of the PCC capability is also checked (as part of instruction fetching). Most other instructions authorise loads and stores via a capability in an explicitly identified register, or use DDC, rather than implicitly use PCC.

Conventional execution flow is also controlled by capabilities, with branch instructions to capability destinations (or implicitly w.r.t. the PCC for legacy instructions). Here too the capability must have its tag set and the target virtual address must be within the bounds, and in this case it must authorise execution.

Then there are instructions to access and manipulate the fields of a capability, including arithmetic on its virtual-address value field (corresponding to conventional pointer arithmetic), comparisons, and other operations to extract and manipulate its permissions and other data.

2.4 Domain Transition

CHERI distinguishes between sealed and unsealed capabilities. An unsealed capability can be used directly (e.g. to load and store), but a sealed capability can only be used to request actions be taken by other software. This feature can be used in the context of *protection domains* or *software compartments*, in which whole subsystems are given access to a limited subset of memory.

Domain X may have no direct authority to domain Y, but may call into domain Y by *invoking* one or more sealed capabilities originally sealed by (or for) Y. The invocation will install unsealed versions of the invoked capabilities in registers. This always includes replacing the current PCC, thus, this performs a jump to a specific code entry point provided by domain Y. These domain transitions are non-monotonic and must be treated specially in our proof.

Variations on this sealing and invocation mechanism enable slightly different calling styles. When sealing capabilities, they can be labelled with an *object type*, if the authorising capability has that object type in its bounds. The "branch to sealed capability pair" instruction invokes a given code capability and also an argument data capability, checking their object types match, providing object-style encapsulation. Three kinds of specialised *sentry* (*sealed entry*) capabilities may

be used transparently by direct branch instructions, memory-indirect branch instructions, and memory-indirect branch-to-pair instructions, respectively.

2.5 Exceptions and the Memory Management Unit

In addition to compiler-facing instructions, system functionality such as virtual memory, cache management, and exception handling is also extended, e.g. adding new exception cause codes, and page-table permission bits for loading or storing capabilities. Because exception handling is able to restore reserved registers during exception-level transitions, it is also a form of domain transition, as reserved registers may contain capabilities not available to the executing code.

2.6 Using CHERI in Software

For context, we sketch how CHERI's capability mechanisms are used by software to control and constrain execution. The CHERI team has adapted a large open-source software stack to CHERI, including the LLVM compiler, linkers, debuggers, multiple OSs, and application suites. The verification in this paper is motivated by this software usage, but is itself purely about the architecture.

One of the main uses of capabilities is fine-grain memory protection. *Spatial memory safety* is achieved in CHERI C/C++ by implementing explicit pointers (those visible in the language, e.g. variables with pointer type) and implied pointers (used by the generated code and runtime, e.g. the stack pointer, PLT entries, and Global Offset Table pointers) with capabilities instead of conventional machine-word integers. These are protected (from corruption or reinjection) by the CHERI tag mechanism and monotonicity, and hence the memory contents they point to are protected, by the capability permissions and bounds checks, so long as no other capabilities give undesired access to them. This relies on compiler-generated code, the kernel, run-time linker, and C runtime (e.g., heap allocator) narrowing capability bounds and permissions during execution as appropriate. This protects against many cases in which a C/C++ coding error could lead to an exploitable vulnerability.

Temporal memory safety, additionally protecting against reuse-after-reallocation errors, is not directly supported by the architecture, but there are a variety of techniques to implement it, especially for heap memory, using CHERI's features [22]. Morello extends the page-table mechanism to allow capability flow to be tracked through memory, supporting revocation of old capabilities.

The other main use of CHERI is *software compartmentalisation*, splitting the address space into different compartments running separate software. The capability monotonicity property ensures these components are contained in their compartment boundaries. Domain transitions are possible via the sealed capability mechanism, which can be used to set up various inter-compartment interfaces. Often these transitions will all be to a privileged control component, but the architecture also supports direct transition between two mutually distrusting pieces of code. Various software models are supported, from implementing fast inter-process IPC to sandboxed libraries within processes.

```
1   function clause __DecodeA64 ((pc, ([bitone,bitzero,bitzero,bitzero,bitzero,bitzero,
2     bitone,bitzero,bitzero,bitzero,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_]
3     as __opcode)) if SEE < 99) = {
4       SEE = 99; let imm17 = Slice(__opcode, 5, 17); let Ct = Slice(__opcode, 0, 5);
5       decode_LDR_C_I_C(imm17, Ct) }
6
7   val decode_LDR_C_I_C : (bits(17), bits(5)) -> unit
8   function decode_LDR_C_I_C (imm17, Ct) = {
9       let 't = UInt(Ct);
10      let offset : bits(64) = SignExtend(imm17 @ 0b0000, 64);
11      execute_LDR_C_I_C(offset, t) }
12
13  val execute_LDR_C_I_C : forall ('t:Int),(0<='t & 't<=31). (bits(64),int('t)) -> unit
14  function execute_LDR_C_I_C (offset, t) = {
15      CheckCapabilitiesEnabled();
16      let base : VirtualAddress = VAFromCapability(PCC);
17      let address : bits(64) = Align(VAddress(base) + offset, CAPABILITY_DBYTES);
18      VACheckAddress(base, address, CAPABILITY_DBYTES, CAP_PERM_LOAD, AccType_NORMAL);
19      data : bits(129) = MemC_read(address, AccType_NORMAL);
20      let data : bits(129) = CapSquashPostLoadCap(data, base);
21      C_set(t) = data }
22
23  val VACheckAddress : forall ('size : Int).
24    (VirtualAddress, bits(64), int('size), bits(64), AccType) -> unit
25  function VACheckAddress (base, addr64, size, requested_perms, acctype) = {
26      c : bits(129) = undefined;
27      if VAIsBits64(base) then { c = DDC_read() }
28      else { c = VAToCapability(base) };
29      __ignore_15 = CheckCapability(c, addr64, size, requested_perms, acctype) }
30
31  val CheckCapability : forall ('size : Int).
32    (bits(129), bits(64), int('size), bits(64), AccType) -> bits(64)
33  function CheckCapability (c, address, size, requested_perms, acctype) = {
34      let el : bits(2) = AArch64_AccessUsesEL(acctype);
35      let 'msbit = AddrTop(address, el);
36      let s1_enabled : bool = AArch64_IsStageOneEnabled(acctype);
37      addressforbounds : bits(64) = address; [...7 lines setting addressforbounds...]
38      fault_type : Fault = Fault_None;
39      if CapIsTagClear(c) then { fault_type = Fault_CapTag }
40      else if CapIsSealed(c) then { fault_type = Fault_CapSeal }
41      else if not_bool(CapCheckPermissions(c, requested_perms))
42          then { fault_type = Fault_CapPerm }
43      else if (requested_perms & CAP_PERM_EXECUTE) != CAP_PERM_NONE
44              & not_bool(CapIsExecutePermitted(c)) then { fault_type = Fault_CapPerm }
45      else if not_bool(CapIsRangeInBounds(c, addressforbounds, size[64 .. 0]))
46          then { fault_type = Fault_CapBounds };
47      if fault_type != Fault_None then {
48          let is_store : bool = CapPermsInclude(requested_perms, CAP_PERM_STORE);
49          let fault : FaultRecord = CapabilityFault(fault_type, acctype, is_store);
50          AArch64_Abort(address, fault) };
51      return(address) }
```

Fig. 2. Sample Morello instruction semantics, in Sail, for parts of the `LDR` (literal) instruction [8, §4.4.76] for loading a capability from a PCC-relative address. Lines 1–5 are the relevant opcode pattern-match clause. That calls the decode function on Lines 7–11, which calls the execute function on Lines 13–21. That uses auxiliary function `VACheckAddress` (Lines 23–29) to check that the PCC capability (wrapped in a `VirtualAddress` structure) has the right bounds and permissions, raising an exception otherwise (Lines 47–50). `MemC_read` (Line 19) performs the load, and `CapSquashPostLoadCap` (Line 20) performs additional checks, in particular clearing the tag of the loaded capability if the authorising capability does not have capability load permission.

3 Concrete Semantics of Morello

The basis for our verification and validation work for Morello is the ISA specification written by Arm in their ASL language. It includes sequential semantics of the capability mechanisms and instructions, along with all of the Armv8-A AArch64 base architecture and its extensions supported by Morello, e.g. floating point and vector instructions, system registers, exceptions, user mode, system mode, hypervisor mode, some debugging features, and virtual memory address translation. In total, the Morello ASL specification is around 62 000 non-whitespace lines, covering 409 instructions, 1050 encodings, 600 automatically generated accessor functions for reading and writing system registers, and 1500 additional helper functions. Arm provided weekly snapshots of the ASL specification while it was being developed.

ASL is a first-order imperative language with exceptions. Originally a paper language only, it was made executable by Reid et al. [40,41]. It supports bitvectors of computed sizes, but bitvector indexing is not statically checked; it also supports mathematical integers and some limited structured types. The Arm documentation provides an informal description of the language [7, Appendix K14], but does not provide a formal semantics. We obtain a formal semantics of Morello by translating the ASL specification into Sail [9], a similar language but with a richer type system and open-source tooling, and thence into Isabelle/HOL, as 90 000 and 210 000 LoS respectively. Fig. 2 shows parts of the Sail semantics for the Morello `LDR` (literal) instruction for loading a capability from a PCC-relative address. This is just an iceberg-tip of the whole semantics, even just for this instruction: the `MemC_read` involves all of address translation, and the call graph of the definitions shown amounts to 7 300 lines of Sail.

We reused the existing open-source Sail tooling and ASL-to-Sail translation [9,10] mostly as-is, with only minor improvements and some engineering work needed to handle Morello. In addition to the Isabelle definitions, we generate a C emulator for validation (§6) using the Sail tool, and we reuse the Isla symbolic execution engine for Sail [10] to generate tests (§7).

4 Abstract Formal Model of Capability Monotonicity

The main challenge in proving whole-ISA security properties of Morello is the scale and complexity of the model. Rather than a direct proof above the 210 000-line Isabelle specification, we factor the proof via an abstraction (instantiated for Morello in §5) that captures the essential properties of arbitrary instruction behaviour in any CHERI ISA. It has to spell out aspects of CHERI in some detail, e.g. the different kinds of non-monotonic domain transitions (cf. §2.4), but it abstracts away ISA details not directly relevant for capability monotonicity.

4.1 ISA Abstraction

The abstraction is defined as properties of an arbitrary sequential ISA semantics, encoded in a monadic type with a trace semantics that exposes the individual

register and memory effects of instructions. This interface was originally designed to connect Sail ISA semantics to relaxed memory models, but we found the factorisation via effects useful for reasoning even in a simple sequential setting.

The monad essentially corresponds to a free monad over an effect datatype. It is parameterised with a return type `'a`, an exception type `'e`, and a sum type of register value types `'regval` (automatically generated by Sail for each ISA):

```
type M 'regval 'a 'e =
  | Done of 'a | Fail of string | Exception of 'e
  | Read_memt of kind * addr * nat * ((bytes * tag) -> M 'regval 'a 'e)
  | Read_reg of register_name * ('regval -> M 'regval 'a 'e)
  ...
```

Finished outcomes either indicate successful termination with a return value `a` (denoted as `Done a`), an exception (`Exception e`) which can be caught using a `try_catch` combinator, or a failure (`Fail msg`), e.g. due to a failed assertion. Effect outcomes carry a continuation that expects a response and returns the next monadic outcome. Monadic `return` wraps a value in `Done`, while `bind` just nests the outcomes without interpreting the effects. We also define a corresponding type of events, e.g. `E_read_reg` (with only concrete values, not continuations), along with an effect trace semantics for monadic expression. We define our requirements on CHERI ISAs in terms of constraints on these traces in §4.4.

4.2 CHERI ISA Parameters

In addition to the ISA semantics themselves, our properties are parameterised on aspects of the ISA relevant to CHERI. This includes names of special registers, in particular the program counter capability register PCC, the invoked data capability register IDC (capability register 29 on Morello, r31 on CHERI-RISC-V), registers holding capabilities to exception handlers (`VBAR_ELn` on Morello), and privileged registers requiring system register access permission.

Moreover, we need to know which instructions may perform sealed capability invocations, as this potentially constitutes a non-monotonic security domain transition. We model this as functions taking an instruction identifier and an effect trace of a particular execution, and returning, respectively, the directly or indirectly invoked sealed capabilities in the trace. For example, the Morello `BRS` instruction invokes the sealed capabilities in its two input registers, and other branch instructions can also invoke sealed capabilities if they are sentries.

Finally, the mapping from virtual to physical memory addresses is captured by a pure partial function taking a virtual address and a (partial) instruction execution trace, from which it can extract the required information about the address mapping to determine the physical address, if any. This is needed because capabilities are in terms of virtual addresses, but the memory effects produced by the ISA semantics are in terms of physical addresses, so we need a way to translate between those when formulating requirements on memory accesses in the abstract model. We also assume another function as a parameter to distinguish memory operations that happen as part of an in-memory translation table walk, as the constraints on them differ from those on other memory operations.

4.3 Capability Abstraction

We capture capabilities in the abstract model via a typeclass that provides methods for accessing the various fields of capabilities, as well as sealing and unsealing operations. We also define a notion of *derivability* that serves as an upper bound on the capability manipulations that instructions are normally allowed to perform. Starting from a set of capabilities C, e.g. provided as inputs to an instruction, the set of capabilities derivable from C is defined inductively as the smallest set that contains C itself as well as capabilities obtained from other derivable ones via one of the following:

- manipulating an unsealed capability c into c' such that bounds or permissions are not increased, formalised using an ordering where $c' \leq c$ iff either $c' = c$, or c' is untagged, or both are tagged and unsealed and the bounds and permissions of c include those of c';
- turning a capability into a sealed entry capability;
- sealing a capability using another derivable sealing authority capability, setting the object type of the sealed capability to the current address value of the authority capability (interpreted as an object type), if the authorising capability is tagged and unsealed, has sealing permission, and its value (and therefore the object type) is within its bounds; or
- unsealing a capability using another derivable unsealing authority capability, if the latter is tagged and unsealed, has unsealing permission, and its value is within bounds and matches the object type of the sealed capability.

Of these operations, unsealing is the only one that may grant new privileges that are not already granted by the input capabilities. However, unsealing requires specific authority. An operating system, for example, can control what capabilities a user-space process can unseal by only handing out unsealing authority capabilities with a limited set of object types in their bounds.

4.4 CHERI ISA Intra-instruction Properties

Our abstraction is defined as the conjunction of four instruction-local properties. They are relatively straightforward to verify for a concrete ISA, and we will describe the proof for Morello in §5. At the same time, the properties imply the whole-ISA property of reachable capability monotonicity, as explained in §4.5. Hence, they serve as a useful intermediate abstraction layer for structuring the overall proof.

The central security guarantee that CHERI ISAs aim to provide is that software cannot forge capabilities and thereby escalate its privileges. Hence, we require that instructions only produce capabilities via the above derivation rules, except for the effects of well-defined transition mechanisms for switching control to another security domain.

Property 1 (Capability register writes). In any execution trace of a single instruction, for every write of a tagged capability to a register at a given point in the trace, one of the following holds:

1. The capability is derivable from the capabilities that the instruction has available at this point in the trace.
2. The capability is an invoked capability and written to the PCC or IDC register as part of a sealed capability invocation.
3. The capability has been loaded from an exception handler base register and is written to the PCC register as part of raising an ISA exception.

The first case permits the normal operation of instructions, manipulating capabilities according to the above derivability rules. We allow instructions to use their *available capabilities* in these operations, which normally includes capabilities read from registers or loaded from memory up to the given point in the trace, with some exceptions: First, capabilities read from privileged registers are unavailable unless the system access permission is also available, i.e. if a tagged and unsealed capability with that permission has been read from PCC before. Second, we exclude capabilities loaded as part of translation table walks, as those loads are not subject to capability checks (although none of the existing CHERI ISAs attempt to load capabilities during translation table walks). Third, capabilities used in a domain transition, e.g. capabilities loaded from memory as part of an indirect sealed capability invocation, are unavailable for normal operations and handled separately by the other cases of Property 1 as follows.

The sealed capability invocation case applies when the capability being written is an invoked capability of the current instruction, as declared when instantiating the CHERI ISA abstraction (see §4.2). Such an invocation performs a branch to the unsealed code capability by writing it to the PCC register, and possibly writes an unsealed data capability to IDC. One of the following cases must hold, representing the different supported kinds of capability invocation:

Sealed pair A pair of capabilities sealed with the same, non-sentry object type and with BranchSealedPair permission is available, the capability that is being written is an unsealed version of one of those, and it is written either to PCC and it has the execute permission, or it is written to the invoked data capability register IDC and does not have the execute permission.

Direct sentry The capability is written to PCC, and a version of it that is sealed with a sentry object type is available to the instruction.

Indirect sentry An indirect sentry capability is available and used to load either two capabilities from memory that may be written to the PCC and IDC registers, or one capability that may be written to PCC while the unsealed version of the indirect sentry itself may be written to IDC.

The ISA exception case is signalled in the Morello model by the helper function `AArch64.TakeException` throwing a (Sail language) exception after setting up the branch to the exception handler. In this case, we allow a capability to the exception handler to be read from a privileged exception handler base register and written to PCC, even if system register access permission is not available. However, the definition of available capabilities together with our properties guarantee that this capability is not used for any other operations.

```
let store_cap_reg_axiom ISA has_ex invoked_caps invoked_indirect_caps t =
  let use_mem_caps = (invoked_indirect_caps = {}) in
 (∀ i c r. (writes_to_reg_at_idx i t = Just r ∧ c ∈ (writes_reg_caps_at_idx ISA i t))
    ⟶
    (∗ Only store monotonically derivable capabilities to registers ∗)
    (cap_derivable (available_caps ISA use_mem_caps i t) c ∨
    (∗ ... or perform one of the following non − monotonic register writes : ∗)
    (∗ Exception ∗)
    (has_ex ∧ c ∈ exception_targets_at_idx ISA i t ∧ r ∈ ISA.PCC) ∨
    (∗ Capability pair invocation ∗)
   (∃ cc cd. ((c ≤ (unseal cc) ∧ r ∈ ISA.PCC) ∨ (c ≤ (unseal cd) ∧ r ∈ ISA.IDC)) ∧
      cap_derivable (available_caps ISA use_mem_caps i t) cc ∧
      cap_derivable (available_caps ISA use_mem_caps i t) cd ∧
      invokable cc cd ∧ c ∈ invoked_caps) ∨
    (∗ Direct sentry invocation ∗)
    (∃ cs. c ≤ (unseal cs) ∧ is_sentry cs ∧ is_sealed cs ∧ r ∈ ISA.PCC ∧
      cap_derivable (available_caps ISA use_mem_caps i t) cs ∧
      c ∈ invoked_caps) ∨
    (∗ Indirect sentry invocation (writing the unsealed sentry to IDC) ∗)
    (∃ cs. c ≤ (unseal cs) ∧ r ∈ ISA.IDC ∧ is_indirect_sentry cs ∧ is_sealed cs ∧
      cap_derivable (available_reg_caps ISA i t) cs ∧
      c ∈ invoked_indirect_caps) ∨
    (∗ Indirect capability (pair) invocation ∗)
    (∗ (writing the loaded capability/capabilities to PCC/IDC) ∗)
    (∃ c'. ((c ≤ (unseal c') ∧ is_sealed c' ∧ is_sentry c' ∧ r ∈ ISA.PCC) ∨
              (c ≤ c' ∧ r ∈ (ISA.PCC ∪ ISA.IDC))) ∧
      cap_derivable (available_mem_caps ISA i t) c' ∧
      c ∈ invoked_caps ∧ invoked_indirect_caps ≠ {})))
```

Fig. 3. Formal definition of capability register write Property 1, slightly simplified

We formalise Property 1 as a predicate on traces, given in Fig. 3. It takes a number of arguments that we instantiate using the CHERI ISA parameters of §4.2, e.g. with *invoked_caps* set to the capabilities that the given instruction invokes in the given trace. The predicate details the different cases (and invocation subcases) of Property 1 for all capabilities written to registers, using helper definitions such as available_caps or invokable (checking permissions and object types of a pair of sealed capabilities).

The other three properties state that capabilities stored to memory must be derivable from available capabilities (here there are no non-monotonic exception cases), and that accesses to memory or privileged registers must be authorised by capabilities with sufficient permissions and bounds.

Property 2 (Capability stores). Every tagged capability stored to memory at a given point in an execution trace of a single instruction is derivable from the available capabilities at that point in the trace.

Property 3 (Privileged registers). Reads from or writes to privileged registers in an execution trace of a single instruction happen only after a tagged and unsealed capability with system register access permission has been read from PCC, unless an ISA exception is raised in the trace and the event is a read from an exception handler base register.

Property 4 (Memory accesses). For every load or store event at a given point in an execution trace of a single instruction, there is a tagged capability available at that point in the trace that authorises the memory operation (further explained below), unless the event is part of a translation table walk. The authorising capability must be unsealed, unless it is an indirect sentry capability being invoked in this trace and the event is a load. If the event is a load or a store of a tagged capability, then the address must be aligned to the capability size.

The authorising capability for memory accesses must be tagged and have the right bounds and permissions: the latter must include load/store permission, and there must be a virtual address range covered by the bounds of the capability that translates to the physical address range covered by the memory event. Loading/storing capabilities (and not just untagged data) requires additional permission bits. The authorising capability must also normally be unsealed; the only allowed case of using a sealed capability for a memory operation is the invocation of an indirect sentry capability. In that case, Property 1 allows the loaded capability (or pair of capabilities) to be written to PCC (or IDC). However, due to the definition of available capabilities, the loaded capabilities will in this case be unavailable for other purposes. Only capabilities loaded via unsealed authorising capabilities can be used for regular operations.

In addition to the instruction semantics, our ISA models also contain ASL/Sail code defining instruction fetch and decode behaviour. We use this for generating emulators, but also for stating the whole-ISA monotonicity theorem below with respect to multi-instruction traces produced by a fetch-decode-execute loop. For the fetch segments of these traces, we require the same properties to hold as for individual instruction execution traces, with the only difference being in the authorisation of memory loads: we assume that instruction fetching only loads instructions from memory, so we do not allow instruction fetching to perform capability memory loads, and we require that it checks for the execute rather than the load permission in the authorising capability.

4.5 Capability Monotonicity Theorem

The above single-instruction properties are sufficient to prove a whole-ISA monotonicity theorem for *reachable capabilities*. This set of reachable capabilities for a given state of the system is defined inductively as the smallest set that includes:

- capabilities in non-privileged registers, and those in privileged registers if a tagged and unsealed capability with system access permission is reachable;
- in-memory capabilities at capability-aligned virtual addresses, if there is a reachable capability that authorises loading the capability; and
- capabilities derivable from reachable capabilities via the rules of §4.3, i.e. restricting bounds or permissions, creating sentry capabilities, or sealing/unsealing capabilities (if a suitable authorising capability is also reachable).

This set is intended to provide an upper bound on the set of capabilities that software can construct (on its own) when starting execution in the given state, and the monotonicity theorem confirms that it is indeed an upper bound.

We assume a sequential setting and state the theorem with respect to executions of a sequential fetch-decode-execute loop; reasoning about concurrent behaviour is beyond the scope of this paper. Executing an effect trace t from a state s leading to a state s', written $s \xrightarrow{t} s'$, is possible if the register and memory contents in read events along the trace t correspond to the last written values, if any, or the contents in the initial state s otherwise, and if s' results from s by updating register and memory contents with the values in t.

Proving the instruction-local properties of the last subsection for a concrete ISA might also require certain architecture-specific assumptions. We allow the specification of both a capability invariant that is preserved by capability derivation and assumed to hold initially, and a predicate on traces capturing further assumptions, e.g. about system registers. We say that an architecture is a *CHERI ISA* if all possible traces of instruction execution and fetching that satisfy the architecture-specific trace assumptions, and that read only capabilities satisfying the architecture-specific capability invariants, satisfy the properties of §4.4. Reachable capability monotonicity then holds for executions of arbitrary sequences of instructions, unless and until a transition to another security domain occurs via an ISA exception or sealed capability invocation.

Theorem 1 (Reachable Capability Monotonicity). *Let $t = tf_1 \cdot te_1 \cdot tf_2 \cdot te_2 \cdot \ldots$ be a trace of the fetch-decode-execute loop of a CHERI ISA, alternating fetch/decode traces tf_i and instruction execution traces te_i, and let s be a state such that $s \xrightarrow{t} s'$. If all of the following hold:*

1. *all traces tf_i and te_i satisfy the architecture-specific assumptions,*
2. *the capabilities in s satisfy the architecture-specific capability invariants,*
3. *none of the fetch and execute traces tf_i and te_i raise an ISA exception,*
4. *the address translation mapping stays invariant along t, and*
5. *unsealed versions of the invoked sealed capabilities in t are reachable in s,*

the set of capabilities reachable in s' is a subset of the capabilities reachable in s.

This guarantees that software cannot escalate its privileges by forging capabilities that are not reachable from the starting state. Non-monotonic changes in the set of reachable capabilities are limited to the specific mechanisms defined above for transferring control to another security domain, i.e. ISA exceptions or sealed capability invocations, installing capabilities belonging to the new domain in the PCC (and possibly IDC) register. The monotonicity guarantee stops before such a domain transition happens. Sealed capability invocations within a security domain are monotonic, however; the theorem does cover capability invocation instructions, e.g. branch instructions taking sentry capabilities, if the unsealed invoked capability is reachable in the current security domain (condition 5 above). The translation invariance assumption (condition 4) rules out non-monotonicity due to the interpretation of capabilities changing when the memory mapping changes. It is assumed to hold for the duration of the given intra-domain trace, but after a domain transition and return, e.g. a system call, one could continue using this theorem with a modified translation mapping.

The proof of Theorem 1 starts with an induction on the number of instructions in the trace. For each individual subtrace t of an instruction fetch or execution with $s \xrightarrow{t} s'$, we show that the available capabilities at any point in t are reachable in s, as the definition of available capabilities excludes non-monotonic cases and only includes capabilities that are accessed with suitable permission due to the properties we require. Hence, state updates along t leading to s' (only writing available or invoked, but reachable capabilities due to the requirements and assumptions) are monotonic.

5 Proof of Capability Monotonicity in Morello

5.1 Instantiation of the Abstract Model

In order to instantiate Theorem 1 for Morello, we instantiate the parameters of the abstract model, e.g. the set of privileged registers or the concrete capability representation. We do not currently instantiate the address translation mapping, effectively treating address translation as a black box and assuming an arbitrary but fixed partial mapping, together with a predicate on events to capture assumptions on register and memory contents, under which the mapping produced by the ASL address translation code is guaranteed to coincide with the given mapping. A candidate for instantiating this is the purely functional characterisation of address translation presented in [9, §8] and proved correct there for the base Armv8.3 architecture, under some assumptions about control registers. Using this would also allow (and require) us to substantiate the translation invariance assumption of Theorem 1. In particular, since the translation control registers are protected by the system register access permission, code running without that permission and without write access to the in-memory translation tables cannot modify the translation mapping.

For the monotonicity proof, the main architecture-specific assumption we make is that two privileged system features that could be used to violate monotonicity are inactive: external debuggers, and the experimental instructions `SCTAG` and `STCT` that allow setting tags of arbitrary capability bit patterns. Hence, we make assumptions on the contents of certain control registers to disable these (e.g. `EDSCR.STATUS = 2` to model non-debug state); the tag setting instructions can also be disabled by removing the system access permission.

The capability invariant that we assume in the initial state is that bounds do not go beyond the 64-bit address space and that their length is non-negative, e.g. to rule out memory accesses that wrap around the edge of the address space. There exist capability encodings that violate this property, but the only way to generate them on Morello is via the tag setting instructions or an external debugger, which we assume to be disabled.

We also assume that the PCC capability is initially unsealed, if it is tagged, which the ASL code relies on in a few places. We proved this as an invariant after a bug we found in a branching helper function (see §5.4) was fixed.

Finally, we have to limit certain kinds of "constrained unpredictable" behaviour. For example, the `LDP` instruction loads a pair of words into two desti-

nation registers. However, if the same register index is used for both destination register arguments to the instruction, then it is left underspecified what value is written to the destination register, if any. One might expect this to be either the original register value or one of the loaded values, but Morello inherits from the base Armv8-A architecture the specification that the register value may be set to an architecturally `UNKNOWN` value in such cases. For capabilities, the Morello specification [8] further constrains this in rule TSNJF: "If an `UNKNOWN` value is written to a capability register or to capability-tagged memory, the write does not increase the Capability defined rights available to software." We formalise this by adding an assumption that, in traces for which we want to use the monotonicity theorem, all `UNKNOWN` capabilities used (appearing in traces in nondeterministic choice events) are reachable from the initial state of the trace.

5.2 Manual Proofs about Capability Encoding Functions

We have to prove that the various functions that make changes to the concrete 129-bit capability representation (as used by the instruction semantics) do so in a monotonic way. The challenging aspect is the compressed capability bounds encoding introduced in [58] and used by Morello (as opposed to the version of CHERI-MIPS targeted by previous verification work [38], which used a simpler, uncompressed 256+1-bit encoding). The compression scheme allows the capability address value and both bounds, three 64-bit values, to be encoded in less than 128 bits. This exploits the fact that in well-behaved code the address should be within the bounds or nearby, so the bounds can be expressed as smaller offsets from it. They are encoded in a floating-point style, with an exponent and a floating "mantissa" window. Typical smaller capabilities have precise bounds, but large capabilities require aligned bounds, to save encoding space; the encoding uses various optimisations to maximise precision [58], [8, §2.5.1].

We initially SMT-checked the encoding functions using Sail's existing SMT backend. This provided early design feedback, including discovering an issue in the `CapSetBounds` function (see §5.4).

When moving from SMT checks to Isabelle proofs that can be integrated into the overall proof, one challenging function is `CapIsRepresentableFast`, which checks that an update to the capability value by an offset does not change the decoding of the bounds. It is important for performance that this check is done quickly. This fast version only considers the offset arithmetic within the mantissa window, making pessimistic assumptions about overflow/underflow in lower bits. We can prove that this check is sufficient, using algebraic methods in Isabelle/HOL without bit-blasting or SMT proofs.

The most challenging function for us to verify is called `CapSetBounds`, and is used to narrow capability bounds. The function checks that the requested new bounds fit monotonically in the existing bounds. It also picks an appropriate exponent, aligns to that exponent, and encodes an updated capability.

The main complication is that aligning the bounds to an exponent changes the length slightly, which may be an increase that requires a higher exponent.

The core argument for monotonicity here is non-trivial: the chosen alignment is the minimum one for which bounds can be encoded which enclose the requested bounds. Since the original capability also enclosed this range, its alignment cannot be less than this minimum, thus the bounds of the original capability are already aligned to the selected exponent. This finally implies that coercing the requested bounds to the selected exponent does not move them across the original bounds. A part of the proof of this lemma involved a brute-force split into cases for all possible selected exponents and reducing the cases to SMT bitvector lemmas which we pass to the CVC4 SMT solver [11]. This relies on the solver as an oracle, as replay of bitvector proofs in Isabelle is only experimental. Initial work on the CHERI compression scheme [58] included HOL4 proofs about these two functions, but this is the first time the crucial monotonicity proof has been done for the set-bounds function.

5.3 Proof Engineering

With the model instantiation and lemmas about auxiliary functions in place, the remaining task is to prove that the rest of the ISA uses these functions correctly and satisfies the properties defined in §4.4. We tackle this using a combination of custom proof tactics within Isabelle and an external tool that automatically generates lemmas about the functions and instructions in the architecture. This simple approach worked sufficiently well that we were able to keep up with weekly snapshots of the ASL specification while it was being developed. Re-running the lemma generation tool mostly worked without affecting the existing manually written parts of the proof, with only few exceptions, e.g. when a refactoring of the (crucial) `VACheckAddress` function broke some lemmas about it.

The generated lemmas are stated in terms of predicates that reformulate the properties of §4.4 into properties of partial traces, taking an additional parameter that summarises the capabilities available at the start of this part of the trace. This allows us to split up an instruction proof into proofs that the auxiliary functions satisfy the properties and that they are used correctly, e.g. that a function performing a memory store is only called if a suitable authorising capability is available. Most of these proofs are automatically handled by straightforward proof tactics, but our tooling allows manually overriding specific parts of generated lemmas where necessary. We do this for about 100 of the ASL functions and instructions, generally taking the form of small patches, e.g. giving additional hints to the proof tactics, such as additional simplification rules or loop invariants, or adding side conditions to lemma statements, such as assumptions about capability checks for memory-accessing helper functions. The tool outputs the generated lemmas in theory files which are then checked by Isabelle; hence, the external tool does not need to be trusted. The proof consists of around 37 000 generated lines, 8 600 manually written lines, as well as 8 900 lines for the abstract model, monotonicity proof, and proof tools. The proof executes in 7hrs 20mins CPU time on an i7-10510U CPU at 1.80GHz, but only 3hrs 23mins real time thanks to parallel execution, with peak memory consumption of 18GB.

5.4 Bugs and Issues Found

Our verification work uncovered several bugs and issues in the ASL specification.

During our initial SMT-checking of the capability manipulation helper functions, one issue we discovered that was not known previously was a bug in the top-byte normalisation logic of the `CapSetBounds` function, which could have led to some of the top bits of the lower or upper bound of a capability changing when modifying some of their lower bits, even if the requested bounds were within the original bounds of the input capability, thereby violating monotonicity.

Our Isabelle proof uncovered a bug in the `BranchToCapability` function where the branch target capability was modified without a check that it is unsealed. Hence, branch instructions could have modified sealed capabilities. The result would not have been directly available to the code that performed the branch, because the modified sealed capability would be installed into PCC, and the subsequent instruction fetch would fault with a sealed capability exception, but as part of exception handling the modified sealed capability would then have been written to the CELR register and become accessible to the exception handler.

Another issue we found was a case of missing capability checks in the implementation of the `DC ZVA` instruction. This would have allowed software to overwrite memory regions with zeros without capability authorisation.

We also found various issues that were already known to Arm, e.g. the `STP` instruction checking the tag of the wrong capability, as well as functional bugs not directly affecting our proof of security properties, e.g. a bug in the `LDNP` and `STNP` instructions where the wrong memory access type was used.

We reported all of our findings to Arm, and the issues have been fixed.

6 Validating the Concrete Semantics

Confidence in our results about Morello's security properties relies on our translation of the specification (from ASL into Sail and Isabelle) accurately reflecting the intended architecture. A key part of ensuring that hardware designs implement Arm architectures correctly is to test against Arm's internal Architectural Compliance Kit (ACK); to validate our translation we ran a large collection of tests from the Morello ACK against a Sail generated C emulator. This approach was also taken with an earlier AArch64 Sail model [9]. These tests are typically self-contained executables that can be run directly after processor reset without an operating system or peripherals, except for a simple serial device for reporting results and diagnostic information. Each test executes tens or even hundreds of thousands of instructions, so using our fast C emulator was essential.

The ACK covers Morello-specific functionality alongside the relevant parts of the base Arm-v8.2 architecture in more than 25000 tests. Its scope is wider than the ASL model, including features such as performance counters, debug, and tracing, where the ASL has only interfaces or partial information, leaving the detailed specification to prose descriptions. There are also tests for the generic interrupt controller (GIC), a distinct system-on-chip component with a separate

specification which is not part of the ISA. Moreover, for the Morello-feature suites, the "implementation defined" behaviour expected by the tests is more constrained than normal to match the single Morello hardware design.

To manage this complexity we first obtained baseline results from a Morello Arm Fast Model simulator, without the additional support normally used in the ACK testing environment. This matches the contents of the ASL specification more closely. We then excluded tests which required features that are not fully modelled, and adjusted the "implementation defined" portions of the specification to approximate the hardware. By comparing the results from our Sail generated emulator against the baseline we could identify and repair faults in both the ASL specification and our translation. Repairing these issues was important both to ensure that our understanding of the problem was correct and to ensure that tests could run to completion to rule out further issues.

Specific issues that we encountered involved minutiae about how system register bits behave when features are not present (such as AArch32 instructions), a couple of missing cases in our built-in operations used by SIMD instructions, a variable shadowing issue in our translation tools, corner cases in the ASL specification handling of page table capability tracking, and a few exception handling problems. None of these issues affect capability monotonicity.

The resulting pass rate was 98.1% compared with the baseline. The discrepancies were mostly due to limitations of the ASL model, such as limited debugging support, corner cases in address space handling, and the lack of secure memory; a few details with some SIMD instructions and particular processor exceptions require further investigation, but again, they do not affect monotonicity.

7 Model-based Test Generation

In addition to the ACK, and before we had access to it, we generated a test suite from the model to check core instruction and capability functions against the implementations; and also to adapt QEMU to support most of Morello. We use symbolic execution, well-established as a way to generate high coverage test suites [12,43] and used previously for a much simpler CHERI architecture [13], both to perturb the initial state to explore different instruction behaviours and to control whether processor exceptions are taken. The latter is particularly useful for CHERI ISAs because most input values would trivially fault at one of the capability checks (e.g. see `CheckCapability` in Fig. 2). Instruction set specifications are good candidates for symbolic execution because the languages tend to be relatively simple and the number of paths for any given instruction is bounded. To build a test generator for Morello we were able to reuse the Isla symbolic execution tool, which was already being developed for work combining Sail ISAs with relaxed memory models [10].

The test generator operates on traces of instructions, partially or fully chosen at random from the encoding diagrams included in the original ASL. Isla's symbolic execution was extended with a simple sequential memory model using SMT arrays for the main memory and tags. In outline, the generator: 1. initialises the

model by running the processor reset function in the symbolic executor (this is deterministic and does not involve any symbolic state); 2. alters the state so that the parts the test harness can change are symbolic, and fix other values as necessary (e.g., for memory translation); 3. symbolically executes each instruction in turn to find feasible behaviours and pick one; 4. passes the accumulated path conditions to the Z3 SMT solver [16] to find suitable concrete values for the initial and final states; and 5. constructs the final test with the instructions and the test harness which will set up the initial state and check the final state after execution. This harness is hand-written (although automatically producing it in the style of Martignoni et al. [29] would be interesting to explore), so to accelerate development we first restricted our attention to fault-free behaviours with memory management turned off, then gradually added support for exceptions, for a simple fixed memory mapping, and checks of more of the processor state after execution.

Our coverage goal for test generation was to ensure that all of the specification code for manipulating capabilities and for instructions that were added or modified for Morello would be executed in some test. This was complicated by non-determinism in parts of the specification. Some instructions have "constrained unpredictable" forms which can have one of several effects; e.g., a load-pair where both destination registers are the same might write `UNKNOWN` to them, do nothing, or take a fault. In principle allowing for all of these is possible, but the resulting disjunctions are likely to be much more difficult to solve, and the behaviours themselves are not very interesting, so we discarded these paths.

Another area of non-determinism in the specification is the load/store exclusive instructions that are used for synchronisation. Even during single-core execution these instructions have such behaviour due to the particular memory architecture choices, which are left as unimplemented primitive operations in the specification. To test these instructions we added a simple model of the guaranteed behaviour in Sail, which includes assertions to avoid uncertain cases.

While the number of paths to explore in any instruction is bounded, the number of paths found for some instructions remains impractically large. The main cause is the case splits in the capability compression scheme. We reduce these to a single path by pushing the decisions into the SMT solver using Isla's *linearisation* feature, extended to support more of the language, which transforms functions with no side effects into a single SMT expression. This was sufficient to perform large-scale test generation with the Morello model.

We checked our progress against our coverage goal using the Sail C backend's coverage measurement support, counting, for each expression in a Sail specification, the number of tests that exercise it. Once we had enough tests that the accumulated coverage began to level out, it was apparent that certain instructions and corner cases were not exercised enough. Overriding the random instruction choice filled in most of the gaps, and temporarily disabling the linearisation allowed exhaustive testing of a key capability function.

The tests found a few minor issues in our tooling and some more bugs in the original ASL specification: several undefined variants of instructions were

included, a new load-pair that should have been marked "constrained unpredictable", a set-bounds operation could read the wrong register, and a translation fault could be missed in a load-tags instruction. Corrections were made to the specification for these issues; a couple also arose in one of the implementations of Morello, which were then fixed.

Comparing the coverage of these tests with the ACK is instructive. As we used the Sail coverage as a goal, we hit a few gaps in the ACK, such as the set-bounds issue, and a rare corner case in a core capability function. However, the ACK's coverage goals included semantic notions that we cannot capture easily. For example, if a conditional is supposed to be false because the first of three checks will fail, human-authored coverage includes the other checks passing, whereas our generator does not reason about the other checks because the symbolic execution does not reach them.

The generated test suite was also used as the basis for test-driven development of an extension of QEMU's Armv8-A support to Morello. After adding basics, such as tagged memory and the expanded register file, the tests guided which features to implement, easing development. Small errors were picked up automatically, such as confusing the stack pointer and zero registers (which share an encoding) and sign extension bugs, including one in the pre-existing QEMU code where a previous attempt to fix it had missed a subtle issue.

The adapted QEMU now boots CheriBSD, a version of FreeBSD with capability support, although this required some fixes for issues that were not found by the generated test suite. A few involved parts of the state that were not explicitly included in the self-test, particularly around exception handling, but most of them concerned out-of-scope system features.

8 Related Work

Nienhuis et al. [38] proved similar results for the CHERI-MIPS architecture, above the Isabelle generated from L3 [23]. CHERI-MIPS is much smaller than Morello (6k LoS), and much simpler, without page tables, virtualisation, vector instructions, etc. They identified 9 properties of the ISA semantics that sufficed to show reachable capability monotonicity and a secure encapsulation result. These captured the capability-relevant intentions of instructions explicitly, but were expressed in terms of a conventional whole-system semantics, instead of the intra-instruction semantics we use here, and that was key to scaling. Each instruction had to be annotated with its intention, extensive work was needed to prove commutativity results, and the properties were MIPS-specific.

The other most closely related work, proving properties of capability architectures, establishes stronger results but for highly idealised architecture definitions. While our monotonicity theorem is about arbitrary machine execution up to a domain crossing, Skorstengaard et al. and Georges et al. [46,47,49,48,24] establish logical-relation methods for reasoning about combinations of arbitrary and known code, the latter mechanised in Iris [28], but for idealised machines rather than full architectures. These add new features to help enforcing strong

properties, but with unclear hardware implementation cost. Strydonck et al. [50] and El-Korashy et al. [19] study secure compilation in similarly idealised settings. Ultimately one would like to scale all these methods to production CHERI architectures. de Amorim et al. [5,4] verify information-flow properties of their SAFE architecture, also for a simplified model.

Capabilities have also been used in the interfaces of numerous operating systems. PSOS [37] uses a similar hardware tag bit to CHERI, but all capability operations are implemented in the OS rather than hardware. Various other operating system use standard hardware but have capabilities as part of their interfaces. These systems are very different to CHERI, but their security models have many similarities. Proofs that a (simplified) OS interface matches an abstract capability security model have been done for the EROS OS [45] and for the seL4 kernel [20]. A subsequent proof connects to the seL4 implementation [44]. Each of these abstract models somewhat resembles ours, e.g. with notions of reachable and derivable capabilities. Our observation that domain-crossing events create extra complications also seems to apply to seL4.

There is a great deal of work devoted to other approaches to improve memory safety which we cannot detail here, but see the review [51]. For just a sample, many projects have developed software-implemented variants of C or C++ that provide greater safety, but typically with rather different performance and code-porting costs to CHERI, and without considering whole-system aspects outside a single C/C++ program [25,36,34,35,17,42,21]. Then there are many hardware-accelerated approaches, e.g. MPX and WatchdogLite, Watchdog, and Hardbound [33,32,31,18]. A different line of work aims at bug-finding rather than deterministic mitigation, e.g. AddressSanitizer [2] and many others.

If widely adopted, Morello would radically change the landscape for such work, and for computer security more generally.

Acknowledgements We thank all the members of the wider CHERI and Morello teams, for their work to make Morello a reality. This work was supported by the UK Industrial Strategy Challenge Fund (ISCF) under the Digital Security by Design (DSbD) Programme, to deliver a DSbDtech enabled digital platform (grant 105694), EPSRC programme grant EP/K008528/1 REMS, ERC AdG 789108 ELVER, Arm iCASE awards, EPSRC IAA KTF funds, the Isaac Newton Trust, the UK Higher Education Innovation Fund (HEIF), Thales E-Security, Microsoft Research Cambridge, Arm, Google, Google DeepMind, HP Enterprise, and the Gates Cambridge Trust. Approved for public release; distribution is unlimited. This work was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237 ("CTSRD"), 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 of the Department of Defense or the U.S. Government.

References

1. CHERI. www.cheri-cpu.org (2021), accessed 2021-06-29
2. Sanitizers home page. https://github.com/google/sanitizers (2021), accessed 2021-07-01
3. Morello Sail definitions and proofs. https://github.com/CTSRD-CHERI/sail-morello-proofs (2022)
4. de Amorim, A.A., Collins, N., DeHon, A., Demange, D., Hritcu, C., Pichardie, D., Pierce, B.C., Pollack, R., Tolmach, A.: A verified information-flow architecture. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 165–178. ACM (2014). https://doi.org/10.1145/2535838.2535839
5. de Amorim, A.A., Collins, N., DeHon, A., Demange, D., Hritcu, C., Pichardie, D., Pierce, B.C., Pollack, R., Tolmach, A.: A verified information-flow architecture. J. Comput. Secur. **24**(6), 689–734 (2016). https://doi.org/10.3233/JCS-15784
6. Arm: Arm Morello Program. https://developer.arm.com/architectures/cpu-architecture/a-profile/morello, accessed 2021-06-29
7. Arm: Arm Architecture Reference Manual (Armv8, for Armv8-A architecture profile) (Sep 2017), Arm DDI 0487F.c (ID072120). https://developer.arm.com/documentation/ddi0487/fc/?lang=en. 8248 pages. Accessed 2021-07-02
8. Arm: Arm Architecture Reference Manual Supplement Morello for A-profile Architecture. https://developer.arm.com/documentation/ddi0606/latest (Jun 2021), DDI0606A.j. 1288pp. Accessed 2021-06-29
9. Armstrong, A., Bauereiss, T., Campbell, B., Reid, A., Gray, K.E., Norton, R.M., Mundkur, P., Wassell, M., French, J., Pulte, C., Flur, S., Stark, I., Krishnaswami, N., Sewell, P.: ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In: Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages (Jan 2019). https://doi.org/10.1145/3290384, proc. ACM Program. Lang. 3, POPL, Article 71
10. Armstrong, A., Campbell, B., Simner, B., Pulte, C., Sewell, P.: Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. In: In Proc. 33rd International Conference on Computer-Aided Verification (Jul 2021), extended version available at https://www.cl.cam.ac.uk/~pes20/isla/isla-cav2021-extended.pdf
11. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: International Conference on Computer Aided Verification. pp. 171–177. Springer (2011)
12. Boyer, R.S., Elspas, B., Levitt, K.N.: SELECT—a formal system for testing and debugging programs by symbolic execution. In: Proceedings of the International Conference on Reliable Software. pp. 234–245. ACM, New York, NY, USA (1975). https://doi.org/10.1145/800027.808445
13. Campbell, B., Stark, I.: Extracting behaviour from an executable instruction set model. In: Piskac, R., Talupur, M. (eds.) 2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016. pp. 33–40. IEEE (2016). https://doi.org/10.1109/FMCAD.2016.7886658
14. Chromium: Chromium security. https://www.chromium.org/Home/chromium-security/memory-safety, accessed 2021-06-29

15. Davis, B., Watson, R.N.M., Richardson, A., Neumann, P.G., Moore, S.W., Baldwin, J., Chisnall, D., Clarke, J., Filardo, N.W., Gudka, K., Joannou, A., Laurie, B., Markettos, A.T., Maste, J.E., Mazzinghi, A., Napierala, E.T., Norton, R.M., Roe, M., Sewell, P., Son, S., Woodruff, J.: CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 379–393. ASPLOS '19, ACM (2019). https://doi.org/10.1145/3297858.3304042, https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201904-asplos-cheriabi.pdf
16. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
17. DeLozier, C., Eisenberg, R.A., Nagarakatte, S., Osera, P., Martin, M.M.K., Zdancewic, S.: Ironclad C++: a library-augmented type-safe subset of C++. In: Hosking, A.L., Eugster, P.T., Lopes, C.V. (eds.) Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013. pp. 287–304. ACM (2013). https://doi.org/10.1145/2509136.2509550
18. Devietti, J., Blundell, C., Martin, M.M.K., Zdancewic, S.: Hardbound: architectural support for spatial safety of the C programming language. In: Eggers, S.J., Larus, J.R. (eds.) Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008. pp. 103–114. ACM (2008). https://doi.org/10.1145/1346281.1346295
19. El-Korashy, A., Tsampas, S., Patrignani, M., Devriese, D., Garg, D., Piessens, F.: CapablePtrs: Securely compiling partial programs using the pointers-as-capabilities principle. In: IEEE Symposium on Computer Security Foundations (CSF) (2021)
20. Elkaduwe, D., Klein, G., Elphinstone, K.: Verified protection model of the seL4 microkernel. In: Working Conference on Verified Software: Theories, Tools, and Experiments. pp. 99–114. Springer (2008)
21. Elliott, A.S., Ruef, A., Hicks, M., Tarditi, D.: Checked C: making C safe by extension. In: 2018 IEEE Cybersecurity Development, SecDev 2018, Cambridge, MA, USA, September 30 - October 2, 2018. pp. 53–60. IEEE Computer Society (2018). https://doi.org/10.1109/SecDev.2018.00015
22. Filardo, N.W., Gutstein, B.F., Woodruff, J., Ainsworth, S., Paul-Trifu, L., Davis, B., Xia, H., Napierala, E.T., Richardson, A., Baldwin, J., Chisnall, D., Clarke, J., Gudka, K., Joannou, A., Markettos, A.T., Mazzinghi, A., Norton, R.M., Roe, M., Sewell, P., Son, S., Jones, T.M., Moore, S.W., Neumann, P.G., Watson, R.N.M.: Cornucopia: Temporal Safety for CHERI Heaps. In: Proceedings of the 41st IEEE Symposium on Security and Privacy (SP). pp. 1507–1524. IEEE Computer Society, Los Alamitos, CA, USA (May 2020). https://doi.org/10.1109/SP40000.2020.00098, https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/2020oakland-cornucopia.pdf
23. Fox, A.C.: Directions in ISA specification. In: ITP. pp. 338–344 (2012). https://doi.org/10.1007/978-3-642-32347-8_23
24. Georges, A.L., Guéneau, A., Strydonck, T.V., Timany, A., Trieu, A., Huyghebaert, S., Devriese, D., Birkedal, L.: Efficient and provable local capability revocation using uninitialized capabilities. Proc. ACM Program. Lang. **5**(POPL), 1–30 (2021). https://doi.org/10.1145/3434287

25. Jim, T., Morrisett, J.G., Grossman, D., Hicks, M.W., Cheney, J., Wang, Y.: Cyclone: A safe dialect of C. In: USENIX Annual Technical Conference, General Track. pp. 275–288 (2002)
26. Joannou, A., Woodruff, J., Kovacsics, R., Moore, S.W., Bradbury, A., Xia, H., Watson, R.N.M., Chisnall, D., Roe, M., Davis, B., Napierala, E., Baldwin, J., Gudka, K., Neumann, P.G., Mazzinghi, A., Richardson, A., Son, S., Markettos, A.T.: Efficient tagged memory. In: Proceedings of the 2017 IEEE 35th International Conference on Computer Design (ICCD) (Nov 2017)
27. Joly, N., ElSherei, S., Amar, S.: Security analysis of CHERI ISA. https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/SecurityanalysisofCHERIISA.pdf (Oct 2020), accessed 2021-06-29
28. Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28**, e20 (2018). https://doi.org/10.1017/S0956796818000151
29. Martignoni, L., McCamant, S., Poosankam, P., Song, D., Maniatis, P.: Path-exploration lifting: hi-fi tests for lo-fi emulators. In: Harris, T., Scott, M.L. (eds.) Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012. pp. 337–348. ACM (2012). https://doi.org/10.1145/2150976.2151012
30. Miller, M.: Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. https://github.com/microsoft/MSRC-Security-Research/raw/master/presentations/2019_02_BlueHatIL/2019_01-BlueHatIL-Trends,challenge,andshiftsinsoftwarevulnerabilitymitigation.pdf (Feb 2019), Microsoft Security Response Center (MSRC) BlueHat IL presentation. Accessed 2021-06-29
31. Nagarakatte, S., Martin, M.M.K., Zdancewic, S.: Watchdog: Hardware for safe and secure manual memory management and full memory safety. In: 39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA. pp. 189–200. IEEE Computer Society (2012). https://doi.org/10.1109/ISCA.2012.6237017
32. Nagarakatte, S., Martin, M.M.K., Zdancewic, S.: Hardware-enforced comprehensive memory safety. IEEE Micro **33**(3), 38–47 (2013). https://doi.org/10.1109/MM.2013.26
33. Nagarakatte, S., Martin, M.M.K., Zdancewic, S.: WatchdogLite: Hardware-accelerated compiler-based pointer checking. In: Kaeli, D.R., Moseley, T. (eds.) 12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014. p. 175. ACM (2014), https://dl.acm.org/citation.cfm?id=2544147
34. Nagarakatte, S., Zhao, J., Martin, M.M.K., Zdancewic, S.: SoftBound: highly compatible and complete spatial memory safety for C. In: Hind, M., Diwan, A. (eds.) Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009. pp. 245–258. ACM (2009). https://doi.org/10.1145/1542476.1542504
35. Nagarakatte, S., Zhao, J., Martin, M.M.K., Zdancewic, S.: CETS: compiler enforced temporal safety for C. In: Vitek, J., Lea, D. (eds.) Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010. pp. 31–40. ACM (2010). https://doi.org/10.1145/1806651.1806657
36. Necula, G.C., McPeak, S., Weimer, W.: CCured: Type-safe retrofitting of legacy code. In: ACM SIGPLAN Notices. vol. 37, pp. 128–139. ACM (2002)
37. Neumann, P.G., Feiertag, R.J.: PSOS revisited. In: 19th Annual Computer Security Applications Conference, 2003. Proceedings. pp. 208–216. IEEE (2003)

38. Nienhuis, K., Joannou, A., Bauereiss, T., Fox, A., Roe, M., Campbell, B., Naylor, M., Norton, R.M., Moore, S.W., Neumann, P.G., Stark, I., Watson, R.N.M., Sewell, P.: 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 (SP). pp. 1007–1024 (May 2020). https://doi.org/10.1109/SP40000.2020.00055
39. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer (2012)
40. Reid, A.: Who guards the guards? Formal validation of the Arm v8-M architecture specification. Proceedings of the ACM on Programming Languages **1**(OOPSLA), 88 (2017)
41. Reid, A.: Defining interfaces between hardware and software: Quality and performance. Ph.D. thesis, School of Computing Science, University of Glasgow (March 2019)
42. Ruef, A., Lampropoulos, L., Sweet, I., Tarditi, D., Hicks, M.: Achieving safety incrementally with Checked C. In: Nielson, F., Sands, D. (eds.) Principles of Security and Trust - 8th International Conference, POST 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11426, pp. 76–98. Springer (2019). https://doi.org/10.1007/978-3-030-17138-4_4
43. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Wermelinger, M., Gall, H.C. (eds.) Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005. pp. 263–272. ACM (2005). https://doi.org/10.1145/1081706.1081750
44. Sewell, T., Winwood, S., Gammie, P., Murray, T., Andronick, J., Klein, G.: seL4 enforces integrity. In: International Conference on Interactive Theorem Proving. pp. 325–340. Springer (2011)
45. Shapiro, J.S.: The practical application of a decidable access model. Tech. rep., Citeseer (2003)
46. Skorstengaard, L., Devriese, D., Birkedal, L.: Reasoning about a machine with local capabilities. In: European Symposium on Programming. pp. 475–501. Springer (2018)
47. Skorstengaard, L., Devriese, D., Birkedal, L.: StkTokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities. Proc. ACM Program. Lang. **3**(POPL), 19:1–19:28 (2019). https://doi.org/10.1145/3290332
48. Skorstengaard, L., Devriese, D., Birkedal, L.: Reasoning about a machine with local capabilities: Provably safe stack and return pointer management. ACM Trans. Program. Lang. Syst. **42**(1), 5:1–5:53 (2020). https://doi.org/10.1145/3363519
49. Skorstengaard, L., Devriese, D., Birkedal, L.: StkTokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. J. Funct. Program. **31**, e9 (2021). https://doi.org/10.1017/S095679682100006X
50. Strydonck, T.V., Piessens, F., Devriese, D.: Linear capabilities for fully abstract compilation of separation-logic-verified code. J. Funct. Program. **31**, e6 (2021). https://doi.org/10.1017/S0956796821000022
51. Szekeres, L., Payer, M., Wei, T., Sekar, R.: Eternal war in memory. IEEE Secur. Priv. **12**(3), 45–53 (2014). https://doi.org/10.1109/MSP.2014.44
52. UKRI: Digital security by design. https://www.dsbd.tech/ and https://www.ukri.org/our-work/our-main-funds/industrial-strategy-challenge-fund/artificial-intelligence-and-data-economy/digital-security-by-design-challenge/, accessed 2021-06-29

53. Watson, R.N.M., Laurie, B., Richardson, A.: Assessing the Viability of an Open-Source CHERI Desktop Software Ecosystem. http://www.capabilitieslimited.co.uk/pdfs/20210917-capltd-cheri-desktop-report-version1-FINAL.pdf (September 2021)
54. Watson, R.N.M., Neumann, P.G., Woodruff, J., Roe, M., Almatary, H., Anderson, J., Baldwin, J., Barnes, G., Chisnall, D., Clarke, J., Davis, B., Eisen, L., Filardo, N.W., Grisenthwaite, R., Joannou, A., Laurie, B., Markettos, A.T., Moore, S.W., Murdoch, S.J., Nienhuis, K., Norton, R., Richardson, A., Rugg, P., Sewell, P., Son, S., Xia, H.: Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Tech. Rep. UCAM-CL-TR-951, University of Cambridge, Computer Laboratory (Oct 2020). https://doi.org/10.48456/tr-951, https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-951.pdf
55. Watson, R.N.M., Richardson, A., Davis, B., Baldwin, J., Chisnall, D., Clarke, J., Filardo, N., Moore, S.W., Napierala, E., Sewell, P., Neumann, P.G.: CHERI C/C++ Programming Guide. Tech. Rep. UCAM-CL-TR-947, University of Cambridge, Computer Laboratory (Jun 2020). https://doi.org/10.48456/tr-947, https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-947.pdf
56. Watson, R.N.M., Woodruff, J., Roe, M., Moore, S.W., Neumann, P.G.: Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the Meltdown and Spectre Attacks. Tech. Rep. UCAM-CL-TR-916, University of Cambridge, Computer Laboratory (Feb 2018). https://doi.org/10.48456/tr-916, https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-916.pdf
57. Watson, R.N., Woodruff, J., Neumann, P.G., Moore, S.W., Anderson, J., Chisnall, D., Dave, N., Davis, B., Gudka, K., Laurie, B., et al.: CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In: IEEE Symposium on Security and Privacy (2015)
58. Woodruff, J., Joannou, A., Xia, H., Fox, A., Norton, R., Baureiss, T., Chisnall, D., Davis, B., Gudka, K., Filardo, N.W., Markettos, A.T., Roe, M., Neumann, P.G., Watson, R.N.M., Moore, S.W.: CHERI Concentrate: Practical Compressed Capabilities. IEEE Transactions on Computers **68**(10), 1455–1469 (Oct 2019). https://doi.org/10.1109/TC.2019.2914037, https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/2019tc-cheri-concentrate.pdf