Technical Report

UCAM-CL-TR-959
ISSN 1476-2986

Number 959

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Verified security for the Morello capability-enhanced prototype Arm architecture

Thomas Bauereiss, Brian Campbell,
Thomas Sewell, Alasdair Armstrong,
Lawrence Esswood, Ian Stark, Graeme Barnes,
Robert N. M. Watson, Peter Sewell

September 2021

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

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

This work was partially supported by the UK Government Industrial Strategy Challenge Fund (ISCF) under the Digital Security by Design (DSbD) Programme, to deliver a DSbDtech enabled digital platform (grant 105694), ERC AdG 789108 ELVER, EPSRC programme grant EP/K008528/1 REMS, Arm iCASE awards, EPSRC IAA KTF funding, the Isaac Newton Trust, the UK Higher Education Innovation Fund (HEIF), Thales E-Security, Microsoft Research Cambridge, Arm Limited, 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.

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

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

ISSN 1476-2986

Verified security for the Morello capability-enhanced prototype Arm architecture

THOMAS BAUEREISS, University of Cambridge, UK
BRIAN CAMPBELL, University of Edinburgh, UK
THOMAS SEWELL, University of Cambridge, UK
ALASDAIR ARMSTRONG, University of Cambridge, UK
LAWRENCE ESSWOOD, University of Cambridge, UK
IAN STARK, University of Edinburgh, UK
GRAEME BARNES, Arm Ltd., UK
ROBERT N. M. WATSON, University of Cambridge, UK
PETER SEWELL, University of Cambridge, UK

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 protections it intends to, 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 Morello specification 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 the essential properties of instruction execution in an arbitrary CHERI ISA, 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. We also use Arm's internal test suite to validate our internal 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

1.1 The CHERI and Morello Context

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 [28, 42]. 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, with 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, makes 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 ten 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

Authors' addresses: Thomas Bauereiss, Thomas.Bauereiss@cl.cam.ac.uk, University of Cambridge, Cambridge, UK; Brian Campbell, Brian.Campbell@ed.ac.uk, University of Edinburgh, Edinburgh, UK; Thomas Sewell, Thomas.Sewell@cl.cam.ac.uk, University of Cambridge, Cambridge, UK; Alasdair Armstrong, Alasdair.Armstrong@cl.cam.ac.uk, University of Cambridge, Cambridge, UK; Lawrence Esswood, le277@cam.ac.uk, University of Cambridge, Cambridge, UK; Ian Stark, Ian.Stark@ed.ac.uk, University of Edinburgh, Edinburgh, UK; Graeme Barnes, Graeme.Barnes@arm.com, Arm Ltd., Cambridge, UK; Robert N. M. Watson, Robert.Watson@cl.cam.ac.uk, University of Cambridge, Cambridge, UK; Peter Sewell, Peter.Sewell@cl.cam.ac.uk, University of Cambridge, Cambridge, UK.

Version of 2021-09-06 08:49.

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. This (crucially) 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 [53], 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. 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 [25], and porting the FreeBSD kernel and userspace to CHERI required changes only to 0.18% and 0.04% LoC.

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, to demonstrate viability and enable that pull, and that needs a high-performance silicon processor implementation and software stack above it. 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 [3, 5]. The architecture, emulators, and software toolchains are available already, while physical development boards are expected in 2022; the design has recently taped out. This will allow evaluation of 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.

1.2 Morello Semantics and Verification

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.

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 [4], to which the Morello architecture supplement adds 1200 more [5]. Fortunately, Arm have recently shifted to using an executable version of their ASL language for instruction-set architecture (ISA) specification [38, 40]. 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. 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 [6] language into Isabelle/HOL [37], 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 factorising 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. [36] 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 and feed back to the design team 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 [7], 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 translation 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 Morello architecture, covering the full sequential ISA definition (including all instructions, capability compression, address translation, and so on) (§5).
- Mechanism for auto-generating most of the proof, 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 semantics for Morello can be reused 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 will be made available in due course under a BSD-Clear licence, as will our definitions and proofs.

We begin in §2 by describing the Morello design in more detail, and discuss related work in §8.

Non-goals and limitations. (1) Our results establish confidence that the Morello architecture design satisfies its fundamental intended security properties. This work does not address correctness of the Morello hardware implementation of that architecture, which would be an extremely challenging hardware verification task. (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

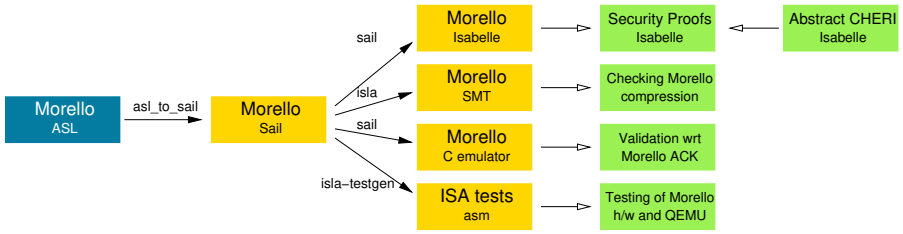


Fig. 1. From Morello ASL source (blue) to auto-generated artifacts (yellow) and verification outcomes (green)

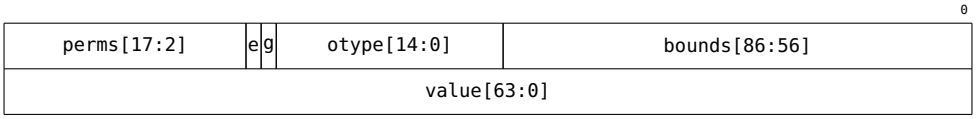
channels, but see [55] for discussion of side-channels and CHERI. (3) We consider only the sequential architecture. Studying concurrency effects would require a more complex system model integrating the Morello sequential semantics with a whole-system concurrency memory model, which we leave to future 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 not 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 [4, Appendix K14], in particular w.r.t. the handling of constrained unpredictable behaviour.

2 OVERVIEW OF THE MORELLO CHERI ARCHITECTURE AND SOFTWARE USE

CHERI is an architectural protection model that extends ISAs with a new data type, the *architectural capability* [53]. 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 [5].

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 circumstances represents a virtual address. The upper 64 bits and tag 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 [5, §2.5.1],[56]. 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. More fine-grained permissions to control loading and storing of capabilities are also available. The System permission controls access to system registers and operations, in addition to the access control mechanisms of the base Arm architecture. Capabilities can be marked as *local* by clearing the Global

permission bit, which stops them from being stored to memory, unless the capability authorising the store has the StoreLocalCapability permission. 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 with an object type requires an authority capability with the Seal (or Unseal) permission and the object type in its bounds; 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. It also extends memory to tagged memory, with a tag bit for each 128-bit sized and aligned unit of DRAM. The Morello hardware provides two implementations of tagged memory: either borrowing bits from the ECC bits of high-end DRAM, or using a tag table and cache mechanism stored in conventional DRAM but associated with memory below the cache subsystem [24]. The two are intended to be indistinguishable architecturally, but differ in their performance and space overheads.

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` [5, §4.4.76]. If the capability does not have its tag set, or the calculated address is outside the bounds for which the current PCC capability authorises reads, a capability fault exception will be raised. Most other instructions load and store integers and capabilities 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 and branch-and-link-register 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.

Due to opcode-space constraints, Morello introduces a new instruction decoding mode that reuses existing integer-relative load/store/jump instructions for capability-relative access.

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 *invoke* mechanism enable slightly different calling styles. 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. Exception handling preserves extended capability register state, and there are new exception cause codes associated with CHERI failures such as bounds violations or untagged memory accesses. Because exception handling is able to restore reserved registers during ring transitions, it is also a form

of domain transition, as reserved registers may contain capabilities not available to the executing code. The page-table format has been extended to add new permissions to, for example, limit loading and storing capabilities.

2.6 Capability-enforced properties

All this design is intended to deterministically enforce various properties. Operations cannot broaden the bounds, expand the permissions, or otherwise expand the rights associated with a capability. Attempts to improperly modify capabilities, or corrupt them in memory via data accesses (e.g. by byte writes), lead to their tag being atomically cleared.

Informally, the architecture aims to enforce [5, §1.2]:

Provenance validity Valid capabilities can only be constructed by instructions that do so explicitly, from other valid capabilities. The hardware provides a universal capability at power-on which can then be subdivided by software.

Capability monotonicity Instructions cannot exceed the permissions and bounds of the original capability when creating new valid capabilities.

Reachable capability monotonicity In any execution of arbitrary code, until execution is yielded to another domain, the set of accessible capabilities cannot increase.

Controlled non-monotonicity Domain-crossing transitions where previously inaccessible capabilities become usable, e.g. on exception entry, are all controlled.

However, these are whole-system properties about arbitrary execution, and it is impossible to check, merely by inspection, that the architecture definition does actually enforce them. Indeed, from the prose statements, one cannot even tell what they mean precisely enough for that to be defined. Our main contribution is to precisely state and prove that the architecture does guarantee reachable capability monotonicity, as given in §4.

2.7 Using CHERI in software

For context, we describe briefly how CHERI's capability mechanisms are used by compiler-generated code, firmware, OS, runtimes, and applications to control and constrain execution. The verification in this paper is motivated by this software usage, but is itself purely about the architecture.

Capabilities can be used in many ways, but the two main usages are fine-grained C/C++ memory protection [11] and scalable software compartmentalisation [54]; the former is currently the most well-developed. A large software stack has been adapted to CHERI and open sourced by the CHERI team, including the LLVM compiler, linkers, debuggers, multiple OSs, and application suites.

2.7.1 Fine-grain memory protection.

Spatial memory safety is achieved in CHERI C/C++ by implementing explicit pointers (those visible in the language, such as variables with pointer type) and implied pointers (those used by the generated code and runtime, such as 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. Here one is relying on compiler-generated code, the kernel, run-time linker, and the C runtime (e.g., heap allocator) to narrow 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 [20]. Morello extends the page-table mechanism to allow capability flow to be tracked through memory, supporting revocation of old capabilities.

2.7.2 Software compartmentalisation. CHERI also allows the address space to be split into software 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.

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

```
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 [5, §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.

instructions, system registers, exceptions, user mode, system mode, hypervisor mode, some debugging features, virtual memory address translation, and a top-level fetch-decode-execute function. Arm provided weekly snapshots of the ASL specification while it was being developed. 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.[1] The snapshots we receive differ

[1]Instructions can be counted in various ways: these counts are of the ASL source definitions, but some of those cover multiple assembly mnemonics and syntactic forms. The Arm documentation groups instructions differently, with Armv8.2-A, upon which Morello is based, having 289 subsubsections in “A64 base instructions” and 375 in “Advanced SIMD and floating-point”, and the Morello supplement listing 161 new instructions, and 173 modified compared to the base architecture.

slightly from the public release of the Morello ASL specification: they contain a top-level fetch-decode-execute function with interrupt handling, as well as some debugging features, default choices for implementation-defined behaviour, and the access logic for system registers in the form of (automatically generated) ASL code rather than XML descriptions.

ASL is a first-order imperative language with exceptions. Originally a paper language only; it was made executable by Reid et al. [38, 40]. 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 [4, Appendix K14], but does not provide a formal semantics. We obtain a formal semantics of Morello by translating the ASL specification into Sail [6], 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. The figure shows 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 and extended the existing open-source Sail tooling for ASL-to-Sail translation and theorem prover definition generation, as well as the C emulator generation and Isla symbolic execution [6, 7]. We wrote a new frontend for the `asl_to_sail` translation tool, using the open-source `asl-interpreter` [39] to parse and typecheck ASL source code, rather than the now-deprecated Arm-internal intermediate representation used in previous work on the Armv8-A ASL specification [6]. We also added heuristics for inferring constraints on integer variables in instruction decoders and propagating them to instruction bodies. For example, many vector instruction decoders calculate the size of elements to process using a left-shift expression like `esize = 8 << scale`, where `scale` has a value between 0 and 3. Propagating the constraint that `'esize in {8, 16, 32, 64}` to an instruction body where this variable is used to determine bitvector lengths is useful both for the Sail typechecker and the backends, e.g. when monomorphising bitvector lengths for generating theorem prover definitions. Otherwise, these tools worked mostly as-is, with only minor improvements needed to handle Morello.

The translation from ASL into Sail and the generation of theorem prover definitions and a C emulator proceed as described in [6]. The `asl_to_sail` tool transliterates the ASL definitions into Sail definitions one by one, runs them through the Sail typechecker, tries to insert runtime assertions in case the typechecking fails due to type constraints that cannot be automatically and statically proved, and asks the user for a manual patch if that fails. For the translation of the Morello ASL into Sail, we provide 32 manual patches to auxiliary functions, mostly adding type annotations, e.g. changing the return type of `CapGetExponent` from `int` to `range(0, 63)`.

The theorem prover backend of Sail translates imperative, effectful Sail definitions into monadic definitions [6, Section 4.2]; we describe the monad used in §4.1. For example, the assignment in Line 35 of Fig. 2 is translated into the monadic bind `(AddrTop address el ≫(λ msbit. ...))` in Isabelle. A non-trivial issue when translating from Sail to Isabelle is that the former supports dependent types to some degree, but the latter does not. This is solved using a bitvector length monomorphisation pass in Sail that duplicates the bodies of functions with dependently typed bitvector lengths, such as vector instructions in Morello that can operate on a range of data sizes, so that the bitvector lengths in each copy can be resolved to fixed values. This relies on the propagation of constraints on integer arguments mentioned above. It also uses heuristics to rewrite some common idioms so that they can be monomorphised more easily, e.g. rewriting some expressions involving variable-width bitvector slices into combinations of shifting and masking operations. We added a few idioms found in Morello to these heuristics, but otherwise did not have to change the Sail tool.

We use all this to generate Isabelle definitions for the full Morello specification, making use of Isabelle's library for fixed-width bitvectors. We also use it to generate a C emulator for validation (§6), and we reuse the Isla symbolic execution engine for Sail [7] to generate tests (§7).

4 ABSTRACT FORMAL MODEL OF CAPABILITY MONOTONICITY

Our goal is to formally verify fundamental security properties of Morello, that instructions correctly perform checks on capability bounds and permissions when accessing resources, and that they do not allow forging capabilities and thereby escalation of privileges. The main challenge here is the scale and complexity of the model. Rather than a direct proof above the 210 000-line Isabelle version of the Morello specification, we factor the proof via an abstract model that captures the essential CHERI properties of arbitrary instruction behaviour, using only a small interface to concrete ISAs in terms of the possible sequences of register and memory effects of individual instructions. We define instruction-local properties of these effect traces, formally capturing fundamental security properties of CHERI ISAs, and prove a whole-ISA capability monotonicity theorem above this abstraction. In this section we present the abstraction, the statement of that theorem, and its proof above the abstraction, and in

§5 discuss the instantiation of the abstraction to Morello, and the proof that the full Morello Isabelle definition satisfies those properties.

We do this specifically for Morello, but the abstraction should also apply to other CHERI ISAs, and preliminary work on CHERI RISC-V suggests it does. The abstraction is really capturing essential properties of an arbitrary CHERI ISA.

4.1 ISA abstraction

The abstraction is defined as properties of an arbitrary sequential ISA semantics. The specification is encoded in a monadic type with a trace semantics that exposes the individual register and memory effects of instructions. This gives us a more fine-grained picture of some large instructions than we would have if the instructions were encoded as a monolithic state update function or similar. In particular, this lets us focus on effects that involve capabilities.

This monadic interface was originally designed, and supported by Sail, to connect ISA semantics to relaxed memory models, where one does not have a simple global memory state. Here, the sequential semantics does have a simple global memory, but the factorisation is useful for reasoning.

The monad essentially corresponds to a free monad over an effect datatype. It is implemented in terms of a type that looks as follows, 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, along with back-and-forth conversions to the individual register value types).

```
type M 'regval 'a 'e =
  | Done of 'a | Fail of string | Exception of 'e
  | Read_memt of read_kind * addr * nat * ((list mem_byte * 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`), a Sail/ASL 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 indicate the kind of effect together with a continuation that expects a response to the effect and returns the next monadic outcome. Monadic `return` wraps a value in `Done`, while `bind` just nests the outcomes without interpreting the effects. As an example, in the prover definitions generated for the functions in Fig. 2, the access to PCC on Line 16 produces a `Read_reg ("PCC", k)` outcome, where the continuation `k` takes a register value for PCC and executes the rest of the function body, i.e. checking the capability and, depending on the result, either producing an `Exception` outcome (as part of `AArch64_Abort` on Line 50) if any of the checks fail, or a tagged memory read outcome `Read_memt` from that address (inside `MemC_read` on Line 19) and ultimately a `Write_reg` outcome to write the loaded and processed capability to the destination register (inside `C_set` on Line 21). We also define a type of events, e.g. `E_read_reg`, corresponding to those outcomes (with only concrete values, not continuations), and define a semantics that assigns the set of possible traces of these effects to each 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:

- the program counter capability register PCC,
- the invoked data capability register (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.

For the latter, we distinguish read- and write-protected registers, as for some, like the hypervisor control registers, we want to ensure that they cannot be written without permission, but we have to allow reads in normal operation to implement their effects on behaviour, e.g. to determine whether to perform a second-stage address translation (these indirect reads of system registers might not be intended to be observable by unprivileged code, but we do not aim to prove such information flow properties in this paper).

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. In the concrete proof for Morello below

we assume an arbitrary but fixed translation mapping, under suitable assumptions on the state. 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 add another function as a parameter to the abstract ISA model to determine which memory operations 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 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 of the authority capability (interpreted as an object type), if the authorising capability is tagged and unsealed, has sealing permission, and its address (and therefore the object type) is within its bounds; or
- unsealing a capability using another derivable unsealing authority capability, if the authorising capability is tagged and unsealed, has unsealing permission, and its address is within bounds and matches the object type of the sealed capability.

Of these capability manipulations, unsealing is the only operation that may grant new privileges that do not already exist in the set of input capabilities. However, unsealing requires specific authority. For example, an operating system can prevent a user-space process from unsealing capabilities with a given object type, by not handing out unsealing authority capabilities with that object type in its bounds.

4.4 CHERI ISA intra-instruction properties

Our abstraction is defined as the conjunction of four properties.

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 produce only capabilities obtainable from their inputs via the derivation rules defined in the previous subsection, 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 program counter or invoked data capability 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 program counter capability register as part of raising an ISA exception.*

The first case permits the normal operation of instructions, manipulating capabilities according to the derivability rules defined in the previous subsection. The *available capabilities* are the set of capabilities that an instruction is allowed to use in these operations, which normally includes capabilities read from registers or loaded from memory before the given point in the trace, except:

- Capabilities read from privileged registers are only considered to be available if sufficient authority for the read is also available, i.e. if a tagged and unsealed capability with the system register access permission has been read from PCC before.
- Capabilities loaded from memory are not considered to be available in some cases. First, invocations of indirect sentry capabilities are handled separately by the second case of the above property, and we want the capabilities indirectly loaded from memory in this case to be only used for the invocation itself, not for regular capability manipulation operations. Hence, we exclude them from the set of available capabilities.[2]

[2]For simplicity we also exclude any other capability loaded in memory in this case, but neither Morello nor other CHERI ISAs have instructions that both perform an indirect sentry invocation and a regular capability load from memory.

Second, capability load instructions in Morello, e.g. the one in Fig. 2, will only check the load capability permission bit (in `CapSquashPostLoadCap`) after performing the load, clearing the tag of the loaded capability if the permission is missing.[3] Hence, we allow the load to proceed (in Property 4 below), but consider the loaded capabilities unavailable.

- Capabilities read from the program counter or invoked data capability registers are only considered to be available if those registers have not been written to before. This serves to rule out a case where an instruction performs a domain transition, i.e. a sealed capability invocation or a jump to an exception handler, by writing capabilities to those registers, but then reads the capabilities back and uses them for other purposes. These capabilities should only be available to the new security domain, starting execution with the next instruction.[4]
- Finally, we consider capabilities appearing in memory load events belonging to a translation table walk to be unavailable. These memory accesses are not protected by capability checks, so we stop any capabilities they load from being used for capability operations (and also require capability stores to be monotonic in Property 3). None of the existing CHERI ISAs attempt to load or store capabilities during translation table walks, however.

Instructions may modify these available capabilities according to the above derivation rules and write the results to registers.

In the sealed capability invocation case, we require that the capability in the register write is declared to be an invoked capability of this execution trace of the given instruction when instantiating the CHERI ISA abstraction (see §4.1). Moreover, one of the following cases must hold, representing the different 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 and does not have the execute permission.

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

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 the invoked data capability (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 our Sail models by throwing a Sail language exception after setting up the branch to the exception handler. In the Morello model, this is done by the `AArch64.TakeException` helper function. In this case, we allow that code to read a capability from a privileged exception handler base register and write it to PCC, even if system register access permission is not available. However, the definition of available capabilities together with the properties of this subsection guarantee that this capability is not used for any other operations.

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 §4.1, 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. For the full definition, we refer to the supplementary material.

Another property we require is that the system register permission is checked when accessing privileged registers, outside of exception handling:

Property 2 (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.*

For capabilities being stored to memory, we have a requirement corresponding to that for capabilities written to registers, but without the exception cases:

Property 3 (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.*

This property is about the contents of memory stores, but we also require that memory accesses (outside translation table walks) happen only if authorised by another available capability:

[3]This function also clears store permissions if the authorising capability does not have the recursive mutable load permission, making the permission handling on loads more fine-grained (see [5, §2.7.4] and [53, Appendix D.3]). We do not currently capture this experimental feature, but it could be handled in a way similar to post-load tag clearing.

[4]Note that in traces generated by the free monad model of ISAs it is not guaranteed that reading a register back after writing will give the same value, but in the sequential model we assume for the whole-ISA monotonicity statement this does hold.

```
let store_cap_reg_axiom ISA has_ex load_caps_permitted invoked_caps invoked_indirect_caps t =
  let use_mem_caps = (invoked_indirect_caps = {} ∧ load_caps_permitted) in
 (∀ i c r.
   (writes_to_reg_at_idx i t = Just r ∧ c ∈ (writes_reg_caps_at_idx ISA i t))
   ⟶
   (∗ Only story 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 ∈ invoked_caps ∧
     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 ∧
     ((leq_cap c (unseal cc) ∧ r ∈ ISA.PCC) ∨
      (leq_cap c (unseal cd) ∧ r ∈ ISA.IDC))) ∨
   (∗ Direct sentry invocation ∗)
   (∃ cs. c ∈ invoked_caps ∧
     cap_derivable (available_caps ISA use_mem_caps i t) cs ∧
     is_sentry cs ∧ is_sealed cs ∧
     leq_cap c (unseal cs) ∧
     r ∈ ISA.PCC) ∨
   (∗ Indirect sentry invocation (writing the unsealed sentry to IDC) ∗)
   (∃ cs. c ∈ invoked_indirect_caps ∧
     cap_derivable (available_reg_caps ISA i t) cs ∧
     is_indirect_sentry cs ∧ is_sealed cs ∧
     leq_cap c (unseal cs) ∧
     r ∈ ISA.IDC) ∨
  (∗ Indirect capability (pair) invocation (writing the loaded capability/capabilities to PCC/IDC) ∗)
   (∃ c'. c ∈ invoked_caps ∧
     invoked_indirect_caps ≠ {} ∧ load_caps_permitted ∧
     cap_derivable (available_mem_caps ISA i t) c' ∧
     ((leq_cap c (unseal c') ∧ is_sealed c' ∧ is_sentry c' ∧ r ∈ ISA.PCC) ∨
      (leq_cap c c' ∧ r ∈ (ISA.PCC ∪ ISA.IDC))))))
```

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

Property 4 (Memory accesses). *For every load or store 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 memory operation is part of an address translation table walk. The authorising capability must be unsealed, unless it is an indirect sentry capability being invoked in this trace and the memory operation 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 must normally be tagged and 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.

The checks that must be performed on the authorising capability in any case, even for indirect sentry invocations, are bounds and permission checks: the permissions must include load/store permission, and there must be a virtual address range covered by the bounds of the capability such that the physical address range covered by the memory event translates to the virtual address range. When loading capabilities, we require the load capability permission bit to be set as well, if we allow the given instruction trace to use capabilities loaded from memory (captured by the *load_caps_permitted* parameter in Fig. 3); otherwise, we allow the load to proceed, but consider the loaded capabilities unavailable, as discussed above. For stores of capabilities, we always require the corresponding permission, as well as an additional permission in case of storing local capabilities.

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, as well as 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 derivation rules defined in §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 and instantiating the monotonicity result 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 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 a transition to another security domain occurs via an ISA exception or sealed capability invocation.

Theorem 4.1 (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 fetch and execute traces tf_i and te_i satisfy the architecture-specific assumptions,*
- *all capabilities reachable in s satisfy the architecture-specific capability invariants,*
- *none of the fetch and execute traces tf_i and te_i raise an ISA exception,*
- *unsealed versions of the invoked sealed capabilities in t are reachable in s, and*
- *the address translation mapping stays invariant along t,*

then 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.

The proof of Theorem 4.1 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

To prove the capability monotonicity theorem for Morello, we first import both the abstract model described in §4 and the Isabelle definitions generated from the Morello ASL specification, as described in §3. The proof covers all parts of the architecture present in the ASL specification (with some broad assumptions about address translation, as discussed below). In this section, we present the theorem, discuss the bugs and issues we found

through the proof work, and explain the structure of the proof, including manual proofs of lemmas about key auxiliary functions as well as automatic proof tools for the bulk of the architecture.

5.1 Instantiation of the abstract model

We instantiate the capability typeclass of the abstract model for the `Capability` type of the Morello specification, which is just a synonym for the type of 129-bit words. The methods of the typeclass are instantiated with the auxiliary functions responsible for accessing or modifying capability fields, such as `CapGetPermissions`, `CapGetBounds`, or `CapUnseal`.

We define the `VBAR_ELn` registers to be the ones containing capabilities for exception handlers, and declare them to be privileged registers protected by the system register access permission for write and (non-exception) read access. In general, there is a broad range of registers which require system register access permission in Morello, but for the purposes of the monotonicity proof we currently focus on capability registers.[5] We do, however, include the (non-capability) address translation control registers, which are protected by the permission bit, in the set of write-privileged registers, as this will be useful when substantiating the address translation assumptions in future work as discussed below. We also include the external debug capability registers `CDBGDTR_EL0` and `CDLR_EL0` in the sets of read/write-privileged registers. These are only accessible in external debug state, but for the monotonicity proof, we assume that no external debugger is active, as this disables some of the capability controls; in particular, system register access permission is implicitly enabled in debug state, and the external debugger can inject arbitrary capabilities. We therefore assume that the system is in non-debug state, i.e. the `EDSCR.STATUS` field has the value 2.

The abstract ISA model requires us to annotate instructions that may perform sealed capability invocations. However, the current version of `asl_to_sail` does not generate an abstract representation of instructions; the `__DecodeExecute` function takes a binary instruction opcode and combines decoding with immediate execution. Implementing a split into separate decode and execute functions with an abstract representation of decoded instructions in between would require some care (in particular, the representation would need a hierarchical structure split into subtrees of instructions, as a flat enumeration of the more than 400 instructions would overwhelm Isabelle's datatype package). Instead, we manually patch the decode clauses of the specific instructions that may perform sealed capability invocations to announce themselves by writing a value representing which invocation instruction it is and what its arguments are (in particular which registers have capabilities being invoked) to a pseudoregister before instruction execution. This allows us to define a function that takes a decode/execute trace and decides whether it is an invocation instruction and which registers it invokes by looking at the first event, rather than duplicating the decoding logic to define a function that determines this information from the binary opcode. We similarly annotate capability load instructions with the information which register they use to get the authorising capability in order to capture the post-load tag clearing discussed in §4.4.

We also patch the top-level step function of the model to clearly separate between the instruction fetch and decode/execute phases, so that we can talk about these phases and the different requirements on them in the monotonicity proof.

For the monotonicity proof, we require a number of architecture-specific assumptions, the first of which is the already mentioned non-debug state assumption. Second, we have to ensure that the experimental privileged capability creation instructions `SCTAG` and `STCT` cannot be used to set the tags of arbitrary capability bit patterns without further checks, as this is inherently non-monotonic. These instructions are generally unavailable to user-space code running at exception level EL0, and for higher exception levels there are two ways to control them: One is via the control register fields `{CHCR_EL2,CSCR_EL3}.SETTAG`, which allow the disabling of tag setting for exception levels below EL3. The other control is via the system register access permission; without this permission in the PCC capability, the `SCTAG` and `STCT` instructions only allow (monotonic) tag clearing, not tag setting. We assume that one of those controls is in place to prevent arbitrary tag setting.

Third, the capability invariant we assume 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. Hence, in the context of the monotonicity theorem, it is an invariant and we assume it for the reachable capabilities in the initial state.

Fourth, another invariant that we assume for the initial state is that the capability in the PCC register is unsealed, if it is tagged. The instruction fetch code in the ASL specification does use the `CheckCapability` function to check for a sealed capability, but it also uses `AArch64.TakeException` in case of exceptions, which requires the PCC capability to be unsealed, as it might modify it. We prove that the unsealedness of PCC is an invariant due to

[5] A configurable Arm feature that we do not currently handle is banking of registers like SP per exception level; this could be handled by treating banked registers of higher exception levels in a way similar to privileged registers.

the fact that the `BranchAddr` helper function called before writing a capability to PCC clears the tag of sealed capabilities.

Fifth, we have to limit certain kinds of "constrained unpredictable" behaviour. For example, the `LDP` instruction loads a pair of words (or capabilities) into two destination registers. However, if the registers overlap, i.e. 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 constrained to 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 [5] 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 as nondeterministic choice events) are reachable from the initial state.

Finally, address translation is currently treated as a black box in the proof: we assume an arbitrary but fixed partial mapping from virtual to physical addresses together with a predicate on events that captures assumptions on register and memory contents such that, for all complete traces of the `AArch64.FullTranslateWithTag` function defined in the concrete Morello model where all events satisfy the assumption predicate, the returned physical address matches the one returned by the abstract mapping. We also assume that the abstract mapping is paged, guaranteeing that aligned memory accesses of the small sizes occurring in the Morello specification do not cross page boundaries. We plan to instantiate the abstract mapping with a purely functional characterisation of address translation like the one presented in [6, Section 8] and proved correct there for the base Armv8.3 architecture, under some assumptions about control registers. This would also allow (and require) us to substantiate the translation-related assumptions. 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. However, at the time of writing, extending the proof in this direction remains future work.

5.2 Manual proofs of lemmas about auxiliary functions

Having instantiated the general CHERI abstraction to Morello via the helper functions `CapGetBounds`, `CapGetPermissions`, etc, we must manually prove a number of consistency proofs. For instance, we must show that the helper check functions like `CheckCapability` really check the right properties, and that memory access as done in Morello agrees with the assumptions we have made about address translation. Most of these proofs are straightforward.

However, proofs about capability encoding are more complex. Morello implements the compressed capability variant of CHERI introduced by Woodruff et al. [56]. This is a limited-precision encoding. To a first approximation, the compressed encoding works by encoding only a floating "mantissa" window of the bits of the upper and lower bound of the capability, and an exponent which positions the mantissa. The bounds are aligned to this exponent, so the bits below the mantissa window are zero. The bits above the mantissa window are taken from the capability value, with a possible +1 or -1 offset. Various tweaks are made to squeeze extra bits in the encoding of this scheme, which are fully explained in Woodruff et al. [56], [5, §2.5.1].

The correctness of the few functions that directly interact with this encoding is of crucial importance for the overall security properties of Morello. We initially SMT-checked key properties of these functions using Sail's existing SMT backend in order to provide quick feedback early on, which did uncover an issue in the `CapSetBounds` function that was reported and fixed (see §5.4). We then proceeded to manually prove correctness properties of these functions in Isabelle/HOL and integrate these properties into the overall capability monotonicity proof.

The first tricky function, `CapIsRepresentableFast`, is used to check that arithmetic on capability values is safe. It is essential that adding offsets to pointers is fast, taking only a single cycle on the Morello CPU. However some bits of the capability value are used in decoding the bounds, so a value change might (non-monotonically) change the bounds. The simple way to prevent this (`CapIsRepresentable`) is to decode the bounds before and after the change and compare them. The faster operation `CapIsRepresentableFast` does a limited-precision version of the addition within the mantissa bits, and checks for a particular variant of overflow that would change the bounds. This addition assumes the pessimistic case for the value carried in from the addition in the lower bits.

We prove that this fast check implies that the bounds are identical. This can be done in Isabelle/HOL in a classic algebraic style without bit-blasting.

The most involved function for us to verify in Morello is called `CapSetBounds`. This function is used to narrow the bounds of a provided capability, by specifying its new lower bound and length. Monotonicity naturally requires that the function narrows bounds and never widens them. There are a number of complications:

- because of the limited precision, the set-bounds function may need to pick an exponent and align the requested bounds to it;

- the bounds are aligned outwards, i.e. the lower bound is aligned down and the upper bound aligned up;
- there is a feedback effect, in which aligning the bounds to some exponent may increase the difference between the bounds and then require a higher exponent;
- the set-bounds function both selects bounds and encodes them into the capability in one function, creating a single large proof problem;
- the encoding and decoding is slightly non-uniform for very large caps, with the possibility that the top of the mantissa is at or above the top bit of the 64-bit address, which requires handling some strange special cases; and
- only the requested bounds, not the finally selected bounds, are compared to the bounds of the original capability.

The bounds widening is provided for use in low-security best-effort cases. Morello, and other CHERI architectures, also provide an "exact" variant of bounds setting which does not widen the bounds. This flexible version is for best-effort cases, for instance when narrowing capability bounds to some user-provided array whose bounds might not be possible to encode exactly.

The Morello specification checks the requested bounds, not the final bounds, because that is how the hardware is expected to work. This allows the check to be done in parallel to calculation of the final bounds. This illustrates a tricky trade-off in writing an architecture specification: is it better to reflect what the hardware does, or what the higher-level design goals are?

The key observation, from which we can argue that this function is safe, is that the set-bounds function selects the tightest aligned bounds that can be encoded in the scheme. Specifically, it picks the best possible alignment. We define the relevant "large enough alignment for bounds" predicate in Isabelle, and show that the original cap's alignment was large enough for its bounds, and thus large enough for the narrower requested bounds, and thus at least as the (best possible) selected alignment. Therefore the original bounds were aligned at least to the selected alignment, and coercing to the final bounds cannot move any further than the original bounds.

Finally, to deal with the complexity of the encoding, we prove a helper lemma relating `CapSetBounds` to its counterpart `CapGetBounds`. We show that, if bounds are set and then decoded, that:

- the decoded bounds are the requested bounds, coerced to the new cap's alignment;
- if the new cap's alignment e has $e > 0$, then $e - 1$ is not a possible wrapping alignment.

We call the above the "brute force lemma" for `CapSetBounds`. To prove this we reduced the problem to bitvector arithmetic in Isabelle, and then passed this problem to the CVC4 SMT solver [8]. Unfortunately replay of bitvector SMT proofs in Isabelle is only experimental, so this proof depends on the SMT solver as an oracle. In addition we had to adjust the SMT settings in Isabelle to use a more recent and stable version of CVC4 than the one that ships with Isabelle2020 by default.

5.3 Proof automation

With the abstract model instantiated and lemmas about the relevant auxiliary functions in place, the remaining task is to prove that the rest of the architecture specification 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. The tool reads a configuration file with information about the architecture, e.g. what the name of the capabilities datatype is and which registers are privileged, and it uses the Sail API to parse, type-check, and analyse the architecture specification. It then generates lemmas about all auxiliary functions and instructions in the architecture, in topological order, to show that they satisfy our properties.

These lemmas are stated in terms of predicates that reformulate the properties of §4.4, originally defined for complete traces of instructions, into properties of partial traces, taking an additional parameter that summarises the capabilities that are 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 after a suitable capability check has been performed. We defined Isabelle proof tactics that prove these subgoals using generic proof rules of the abstract model, auto-generated lemmas, and the manually stated helper lemmas. The tool allows manually overriding specific parts of generated lemmas. 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 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 Isabelle theory files, which get imported by a top-level theory file that instantiates the monotonicity theorem of the abstract model for Morello. 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 (and 1h 15mins on an Intel Xeon W-2145 with 3.70GHz) thanks to parallel execution,

with peak memory consumption of 18GB. While the ASL specification was being developed, we kept up-to-date with the weekly snapshots of the specification by re-running the lemma generation tool. This mostly worked without breaking the existing manually written parts of the proof, with the exception of a few occasions where auxiliary functions about which manually proved lemmas existed where modified or refactored, e.g. when the `VACheckAddress` function was modified to take the address as a separate argument rather than reading it from the input capability.

5.4 Bugs and issues found

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

During our initial SMT-checking of monotonicity properties of the capability manipulation helper functions, one issue we discovered that was not known to Arm at the time 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 (not part of the publicly released ASL specification). This would have allowed software to overwrite memory regions with zeros without capability authorisation.

We also found a number of 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 the Arm development team, and the issues have been fixed.

6 VALIDATING THE CONCRETE SEMANTICS

Confidence in our results about Morello's security properties relies on the specification, and our translation of it 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), and in a similar way 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 [6]. 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 interrupt controller, which is not part of the base architecture at all. For the Morello-feature suites, the tests are much more tailored to the implementation than normal and expect the "implementation defined" behaviour to match the single Morello hardware design.

To manage this complexity we first obtained a baseline from results from a Morello Arm Fast Model simulator, without the additional support normally used in the ACK testing environment. This more closely matches the contents of the ASL specification. 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 remaining tests were checked by hand: most were 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

Having such a detailed model of the architecture, especially in a language with symbolic execution tooling, gave us the opportunity to automatically generate a test suite from the model, complementing Arm's ACK. Before we had access to the ACK, these generated tests were used to check core instruction and capability functions against the implementations, and to adapt QEMU to support most of Morello.

Symbolic execution is well-established as a way to generate high coverage test cases [9, 43]. It has also been used for test generation for a much simpler CHERI architecture [10], 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 architectures because most inputs 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.

The specification language and tooling in that work was different and so could not be directly used here. Fortunately the Isla symbolic execution tool was already being developed for work combining Sail specifications with weak memory models [7], and could be reused as the foundation for building a test generator. Isla further simplifies the Sail language by operating on an intermediate form used in Sail's C backend, and encodes path constraints into SMT formulae so that the Z3 SMT solver [14] can be used to determine whether paths are feasible.

The test generator operates on traces of instruction partially or fully chosen at random from the encoding diagrams included in the original ASL. Isla's symbolic execution is extended with a simple sequential memory model using SMT arrays for the main memory and tag memory. 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 Z3 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.

The test harness is largely hand-written, so to accelerate development we started by restricting our attention to fault-free behaviours with memory management turned off, and gradually added support to switch exception levels, to use a simple fixed memory mapping, and to check more of the processor state after execution. In future, it would be preferable to automatically produce a test harness, perhaps in the style of [27] where small pieces of state manipulating code were automatically sequenced to produce the initial state. Ideally the specification could be exploited to synthesise such pieces of code.

Our coverage goal for test generation was to ensure that all of the expressions in 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 the presence of 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 nonsense to the registers, do nothing, or take an undefined instruction fault. In principle allowing all of these behaviours 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 are 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, which can produce web pages of the original Sail specification annotated by the number of tests that exercise each expression and CSV output showing the accumulated coverage as the number of tests increased. 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 out most of the gaps, and temporarily disabling the linearisation allowed exhaustive testing of a key capability function.

The tests found a few issues in our tooling: mishandling of implicitly declared variables in ASL decoding clauses, and two Sail builtins that were slightly incorrect. More bugs were found 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 corrected.

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. [36] proved similar results for the CHERI-MIPS architecture, above the Isabelle generated from L3 [21]. 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. Our reachable capability monotonicity result is a property about the behaviour of arbitrary code (really, arbitrary machine execution) up to a domain crossing. Skorstengaard et al. [46, 47, 48, 49] and Georges et al. [22] establish logical-relation methods for reasoning about combinations of arbitrary and known code, the latter mechanised in Iris [26], but about idealised capability machines rather than full architectures. These capability machines are extended with new features to make it easier to enforce strong properties, but whose hardware implementation cost is unclear. Strydonck et al. [50] and El-Korashy et al. [17] study secure compilation in similarly idealised settings. Ultimately one would like to scale all these methods to production CHERI architectures. de Amorim et al. [12, 13] 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. In PSOS [35] the capabilities are interpreted by the OS but protected by a hardware tag bit with similarities to CHERI. In most other cases, capabilities are provided entirely by the OS above standard hardware. These differ substantially to hardware capabilities, but the security models that abstract over them have many similarities. Comparisons between an abstract capability security model and an actual OS interface have been done informally for the EROS OS [45], formally (but with a simplified model of the interface) for the seL4 kernel [18], and also formally in a proof linked to the seL4 implementation [44]. Each of these presents an abstract model closely resembling ours, for instance, with a notion of reachable and derivable capabilities. Our observation that domain-crossing events create extra complications also seems to apply to the seL4 implementation.

There is a great deal of work devoted to other approaches to improve memory safety which we cannot detail here, but see the review by Szekeres et al. [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 [15, 19, 23, 32–34, 41]. Then there are many hardware-accelerated approaches, e.g. MPX and WatchdogLite, Watchdog, and Hardbound [16, 29–31]. 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.

ACKNOWLEDGMENTS

This work was partially supported by the UK Government Industrial Strategy Challenge Fund (ISCF) under the Digital Security by Design (DSbD) Programme, to deliver a DSbDtech enabled digital platform (grant 105694), ERC AdG 789108 ELVER, EPSRC programme grant EP/K008528/1 REMS, Arm iCASE awards, EPSRC IAA KTF funding, the Isaac Newton Trust, the UK Higher Education Innovation Fund (HEIF), Thales E-Security, Microsoft Research Cambridge, Arm Limited, 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] 2021. CHERI. www.cheri-cpu.org. Accessed 2021-06-29.

[2] 2021. Sanitizers home page. https://github.com/google/sanitizers. Accessed 2021-07-01.

[3] Arm. [n.d.]. Arm Morello Program. https://developer.arm.com/architectures/cpu-architecture/a-profile/morello. Accessed 2021-06-29.

[4] Arm 2017. *Arm Architecture Reference Manual (Armv8, for Armv8-A architecture profile).* Arm. ARM DDI 0487F.c (ID072120). https://developer.arm.com/documentation/ddi0487/fc/?lang=en. 8248 pages. Accessed 2021-07-02.

[5] Arm. 2021. Arm Architecture Reference manual Supplement Morello for A-profile Architecture. https://developer.arm.com/documentation/ddi0606/latest. DDI0606A.j. 1288pp. Accessed 2021-06-29.

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

[7] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. 2021. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. In *In Proc. 33rd International Conference on Computer-Aided Verification*. Extended version available at https://www.cl.cam.ac.uk/~pes20/isla/isla-cav2021-extended.pdf.

[8] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. Cvc4. In *International Conference on Computer Aided Verification*. Springer, 171–177.

[9] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT—a Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software* (Los Angeles, California). ACM, New York, NY, USA, 234–245. https://doi.org/10.1145/800027.808445

[10] Brian Campbell and Ian Stark. 2016. Extracting behaviour from an executable instruction set model. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, Ruzica Piskac and Muralidhar Talupur (Eds.). IEEE, 33–40. https://doi.org/10.1109/FMCAD.2016.7886658

[11] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. [n.d.]. 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* (New York, NY, USA, 2019-04) *(ASPLOS '19)*. ACM, 379–393. https://doi.org/10.1145/3297858.3304042

[12] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Catalin Hritcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. 2014. A verified information-flow architecture. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 165–178. https://doi.org/10.1145/2535838.2535839

[13] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Catalin Hritcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. 2016. A verified information-flow architecture. *J. Comput. Secur.* 24, 6 (2016), 689–734. https://doi.org/10.3233/JCS-15784

[14] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[15] Christian DeLozier, Richard A. Eisenberg, Santosh Nagarakatte, Peter-Michael Osera, Milo M. K. Martin, and Steve Zdancewic. 2013. Ironclad C++: a library-augmented type-safe subset of c++. In *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*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 287–304. https://doi.org/10.1145/2509136.2509550

[16] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. Hardbound: architectural support for spatial safety of the C programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, Susan J. Eggers and James R. Larus (Eds.). ACM, 103–114. https://doi.org/10.1145/1346281.1346295

[17] Akram El-Korashy, Stelios Tsampas, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. 2021. CapablePtrs: Securely Compiling Partial Programs Using the Pointers-as-Capabilities Principle. In *IEEE Symposium on Computer Security Foundations (CSF)*.

[18] Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. 2008. Verified protection model of the seL4 microkernel. In *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 99–114.

[19] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. 2018. Checked C: Making C Safe by Extension. In *2018 IEEE Cybersecurity Development, SecDev 2018, Cambridge, MA, USA, September 30 - October 2, 2018*. IEEE Computer Society, 53–60. https://doi.org/10.1109/SecDev.2018.00015

[20] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. [n.d.]. Cornucopia: Temporal Safety for CHERI Heaps. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (SP)* (2020-05). IEEE Computer Society, Los Alamitos, CA, USA, 1507–1524. https://doi.org/10.1109/SP40000.2020.00098

[21] Anthony C.J. Fox. 2012. Directions in ISA Specification. In *ITP*. 338–344. https://doi.org/10.1007/978-3-642-32347-8_23

[22] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. 2021. Efficient and provable local capability revocation using uninitialized capabilities. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. https://doi.org/10.1145/3434287

[23] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference, General Track*. 275–288.

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

[25] Nicolas Joly, Saif ElSherei, and Saar Amar. 2020. Security Analysis of CHERI ISA. https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/SecurityanalysisofCHERIISA.pdf. Accessed 2021-06-29.

[26] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[27] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-exploration lifting: hi-fi tests for lo-fi emulators. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, Tim Harris and Michael L. Scott (Eds.). ACM, 337–348. https://doi.org/10.1145/2150976.2151012

[28] Matt Miller. 2019. 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. Microsoft Security Response Center (MSRC) BlueHat IL presentation. Accessed 2021-06-29.

[29] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. 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*. IEEE Computer Society, 189–200. https://doi.org/10.1109/ISCA.2012.6237017

[30] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2013. Hardware-Enforced Comprehensive Memory Safety. *IEEE Micro* 33, 3 (2013), 38–47. https://doi.org/10.1109/MM.2013.26

[31] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2014. WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*, David R. Kaeli and Tipp Moseley (Eds.). ACM, 175. https://dl.acm.org/citation.cfm?id=2544147

[32] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for c. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 245–258. https://doi.org/10.1145/1542476.1542504

[33] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010*, Jan Vitek and Doug Lea (Eds.). ACM, 31–40. https://doi.org/10.1145/1806651.1806657

[34] George C Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe retrofitting of legacy code. In *ACM SIGPLAN Notices*, Vol. 37. ACM, 128–139.

[35] Peter G Neumann and Richard J Feiertag. 2003. PSOS revisited. In *19th Annual Computer Security Applications Conference, 2003. Proceedings.* IEEE, 208–216.

[36] Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. 2020. 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)*. 1007–1024. https://doi.org/10.1109/SP40000.2020.00055

[37] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2012. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Springer.

[38] Alastair Reid. 2017. Who guards the guards? Formal validation of the Arm v8-M architecture specification. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 88.

[39] Alastair Reid. 2019. ASL Interpreter. https://github.com/alastairreid/asl-interpreter.

[40] Alastair Reid. 2019. *Defining interfaces between hardware and software: Quality and performance.* Ph.D. Dissertation. School of Computing Science, University of Glasgow.

[41] Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi, and Michael Hicks. 2019. Achieving Safety Incrementally with Checked C. In *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)*, Flemming Nielson and David Sands (Eds.). Springer, 76–98. https://doi.org/10.1007/978-3-030-17138-4_4

[42] Chromium Security. [n.d.]. https://www.chromium.org/Home/chromium-security/memory-safety. Accessed 2021-06-29.

[43] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *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*, Michel Wermelinger and Harald C. Gall (Eds.). ACM, 263–272. https://doi.org/10.1145/1081706.1081750

[44] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. 2011. seL4 enforces integrity. In *International Conference on Interactive Theorem Proving*. Springer, 325–340.

[45] Jonathan S Shapiro. 2003. *The practical application of a decidable access model.* Technical Report. Citeseer.

[46] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2018. Reasoning About a Machine with Local Capabilities. In *European Symposium on Programming*. Springer, 475–501.

[47] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019. StkTokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *Proc. ACM Program. Lang.* 3, POPL (2019), 19:1–19:28. https://doi.org/10.1145/3290332

[48] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2020. Reasoning about a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management. *ACM Trans. Program. Lang. Syst.* 42, 1 (2020), 5:1–5:53. https://doi.org/10.1145/3363519

[49] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2021. StkTokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *J. Funct. Program.* 31 (2021), e9. https://doi.org/10.1017/S095679682100006X

[50] Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. 2021. Linear capabilities for fully abstract compilation of separation-logic-verified code. *J. Funct. Program.* 31 (2021), e6. https://doi.org/10.1017/S0956796821000022

[51] Laszlo Szekeres, Mathias Payer, Tao Wei, and R. Sekar. 2014. Eternal War in Memory. *IEEE Secur. Priv.* 12, 3 (2014), 45–53. https://doi.org/10.1109/MSP.2014.44

[52] UKRI. [n.d.]. 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] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. 2020. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8).* Technical Report UCAM-CL-TR-951. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-951

[54] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. [n.d.]. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2015-05) *(SP '15)*. IEEE Computer Society, 20–37. https://doi.org/10.1109/SP.2015.9

[55] Robert N. M. Watson, Jonathan Woodruff, Michael Roe, Simon W. Moore, and Peter G. Neumann. 2018. *Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the Meltdown and Spectre Attacks.* Technical Report UCAM-CL-TR-916. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-916

[56] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert Norton, Thomas Baureiss, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel Wesley Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. [n.d.]. CHERI Concentrate: Practical Compressed Capabilities. 68, 10 ([n. d.]), 1455–1469. https://doi.org/10.1109/TC.2019.2914037