Verified security properties and semantics-assisted engineering for the Morello, CHERI-RISC-V, and CHERI-MIPS capability-enhanced architectures

Peter Sewell[1]

TCCoE, 2022-02 (Trusted Computing Center of Excellence Summit)

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

CHERI-MIPS, CHERI-RISC-V: Kyndylan Nienhuis[1], Alexandre Joannou[1], Thomas Bauereiss[1], Anthony Fox[1], Michael Roe[1], Brian Campbell[2], Matthew Naylor[1], Robert M. Norton[1*], Prashanth Mundkur[4], Simon W. Moore[1], Peter G. Neumann[4], Ian Stark[2], Robert N. M. Watson[1], and Peter Sewell[1]

Sail and Isla: Alasdair Armstrong[1], Thomas Bauereiss[1], Brian Campbell[2], Alastair Reid[3*], Kathryn E. Gray[1*], Robert M. Norton[1*], Prashanth Mundkur[4], Mark Wassell[1], Jon French[1*], Christopher Pulte[1], Shaked Flur[1*], Ian Stark[2], Neel Krishnaswami[1], and Peter Sewell[1].

1 University of Cambridge, 2 University of Edinburgh, 3 Arm Ltd, 4 SRI, 5 MPI-SWS, 6 Aarhus University, *when this work was done

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.

CHERI

CHERI: architecture extensions to support *hardware-enforced*

- fine-grain memory protection
- secure encapsulation

using unforgeable capabilities, supporting existing C/C++ code with minimal modification

Academic prototypes CHERI-MIPS, CHERI-RISC-V, and CheriBSD developed since 2010, largely DARPA-funded

CHERI

CHERI: architecture extensions to support *hardware-enforced*

- fine-grain memory protection
- secure encapsulation

using unforgeable capabilities, supporting existing C/C++ code with minimal modification

Academic prototypes CHERI-MIPS, CHERI-RISC-V, and CheriBSD developed since 2010, largely DARPA-funded

Industrial evaluation prototype Morello (CHERI Armv8-A) now a reality:

- architecture specification
- high-performance processor silicon (based on Neoverse N1)
- development board
- software stacks

largely UKRI DSbD and Arm funded

CHERI aims to provide deterministic mitigation of many security vulnerabilities (MSRC estimate 30%–70%).

Potential for mass-market adoption, in Arm A-class cores if Morello experiment works well, and/or in extensions to other architectures – transforming the security landscape.

Questions

But how do we know the architecture *does* enforce the intended security protections?
How can we even *state* them precisely?

(if CHERI is successful, its mechanisms will become a prime target – and if the architecture specification is flawed, all conforming implementations will be – so establishing high confidence is critical)

Questions

But how do we know the architecture *does* enforce the intended security protections? How can we even *state* them precisely?

(if CHERI is successful, its mechanisms will become a prime target – and if the architecture specification is flawed, all conforming implementations will be – so establishing high confidence is critical)

How can we use rigorous semantics in "lightweight formal" ways to ease conventional engineering in the CHERI development process?

Questions

But how do we know the architecture *does* enforce the intended security protections? How can we even *state* them precisely?

(if CHERI is successful, its mechanisms will become a prime target – and if the architecture specification is flawed, all conforming implementations will be – so establishing high confidence is critical)

How can we use rigorous semantics in "lightweight formal" ways to ease conventional engineering in the CHERI development process?

How can we rigorously specify and reason about real-world architectures?

For Morello

Machine-checked mathematical proofs that the fundamental CHERI security properties are guaranteed by the full Morello sequential instruction-set architecture (ISA) specification (62 000 lines, translated via Sail to 210 000 lines of Isabelle). During architecture design, before tape-out.

Model-based test generation, used in pre-Si hardware testing and for QEMU bring-up.

Verified security for the Morello capability-enhanced prototype Arm architecture. Bauereiss, Campbell, T.Sewell, Armstrong, Esswood, Stark, Barnes, Watson, P. Sewell. ESOP 2022.

How did we get here?

CHERI: Hardware/software co-design since 2010:
(Robert N. M. Watson, Simon W. Moore, Peter G. Neumann)

Robert N. M. Watson[1], Simon W. Moore[1], Peter Sewell[1], Peter G. Neumann[7]; at Arm: Graeme Barnes[2], Richard Grisenthwaite[2], Lee Eisen[2], and many more; Hesham Almatary[1], Jonathan Anderson[1*], Alasdair Armstrong[1], Peter Blandford-Baker[1], John Baldwin[7], Hadrien Barrel[1], Thomas Bauereiss[1], Ruslan Bukin[1], Brian Campbell[3], David Chisnall[1*,11], Jessica Clarke[1], Nirav Dave[7*], Brooks Davis[7], Lawrence Esswood[1], Nathaniel W. Filardo[1*,11], Franz Fuchs[4], Khilan Gudka[1*], Brett Gutstein[1], Alexandre Joannou[1], Robert Kovacsics[1*], Ben Laurie[5], A. Theo Markettos[1], J. Edward Maste[1*], Alfredo Mazzinghi[1], Alan Mujumdar[1*], Prashanth Mundkur[7], Steven J. Murdoch[1*], Edward Napierala[4], Kyndylan Nienhuis[1], Robert Norton-Wright[1*,11], Philip Paeps[1*], Lucian Paul-Trifu[1*], Allison Randal[1], Ivan Ribeiro[1], Alex Richardson[1*,5], Michael Roe[1], Colin Rothwell[1*], Peter Rugg[1], Hassen Saidi[7], Thomas Sewell[1], Stacey Son[1*], Ian Stark[3], Domagoj Stolfa[1*], Andrew Turner[1], Munraj Vadera[1*], Jonathan Woodruff[1], Hongyan Xia[1*], Vadim Zaliva[1], Bjoern A. Zeeb[1*],
1 University of Cambridge, 2 Arm, 3 University of Edinburgh, 4 Seoul National University, 5 Google, 6 KAIST, 7 SRI International, 8 University of St. Andrews, 9 Inria Paris, 10 Aarhus University, 11 Microsoft Research, *when this work was done

Architecture semantics (concurrent and sequential) since 2007:

Peter Sewell[1], Susmit Sarkar[1*], Francesco Zappa Nardelli[2*], Scott Owens[1*], Tom Ridge[1*], Thomas Braibant[2*], Magnus O. Myreen[1*], Jade Alglave[2*], Anthony Fox[1*], Samin Ishtiaq[3*], Luc Maranget[2*], Derek Williams[4], Pankaj Pawan[2*], Ohad Kammar[1*], Sela Mador-Haim[5*], Kayvan Memarian[1], Rajeev Alur[5], Milo M. K. Martin[5*], Kathryn E. Gray[1*], Gabriel Kerneis[1*], Dominic P. Mulligan[1*], Christopher Pulte[1], Shaked Flur[1*], Ali Sezgin[1*], Will Deacon[3*], Kyndylan Nienhuis[1], Mark Batty[1*], Jon French[1*], Alasdair Armstrong[1], Thomas Bauereiss[1], Brian Campbell[6], Alastair Reid[3*], Kathryn E. Gray[1*], Robert M. Norton[1*], Prashanth Mundkur[7], Mark Wassell[1*], Ian Stark[6], Neel Krishnaswami[1], Ben Simner[1], Jean Pichon-Pharabod[8], Thibaut Pérami[1], Richard Grisenthwaite[3]
1 University of Cambridge, 2 INRIA, 3 Arm Ltd, 4 IBM, 5 University of Pennsylvania, 6 University of Edinburgh, 7 SRI International, 8 Aarhus University, *when this work was done

CHERI hardware/software/*semantics* co-design since 2014

Morello program since 2019

Focus on the *architectural* abstraction

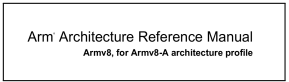
Architecture: central abstraction, specifying the allowed behaviour of hardware, and the basic assumptions for software.

Very attractive from verification p.o.v.:

- relatively stable compared with h/w and s/w implementations
- relatively simple compared with high-level languages and OSs
- relatively trustworthy compared with high-level languages and OSs (hardware vendors do extensive validation, that h/w implements the architecture spec.) (though still defects)
- easier to get confidence that verification relates to what's actually running for the binary

How can we rigorously specify real-world architectures?

Architecture specification – traditional practice



- large (Armv8-A: 8000+ pages) and complex
- sequential behaviour: the *instruction-set architecture* (ISA) fairly well understood (but so much detail!)
- concurrency behaviour: user and system concurrency rather mysterious
- purely prose+pseudocode, not mechanised — so usable only as documentation, not directly for testing, analysis, or proof

Previous formal state of the art?

Sequential ISAs: hand-written semantics for modest fragments
...each verification project rolled its own, e.g.

- Fox for ARM (L3 $\mapsto$ HOL4/Isabelle)
- Hunt et al. for x86 (ACL2)
- Leroy for the assembly abstractions used in CompCert (Coq)
- Benton and Kennedy for x86 fragments (Coq)
- etc.

Concurrency semantics: very unclear

How to express full architecture semantics more formally?

Many existing options, but none support everything we need.

Sail:

- Imperative first-order language for ISA specification (familiar, readable)
- Intentionally limited expressiveness, so very simple semantics; good for analysis
- But enough for full Armv8-A and RISC-V
- Lightweight dependent types for bitvectors (checked using Z3)
- Behaviour of memory actions left to external memory model
  ... so can plug into our semantics and tools for relaxed-memory concurrency
- Open-source tooling to typecheck Sail definitions and generate multiple artifacts...

ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS. Armstrong, Bauereiss, Campbell, Reid, Gray, Norton, Mundkur, Wassell, French, Pulte, Flur, Stark, Krishnaswami, Sewell. In POPL 2019.
Modelling the ARMv8 architecture, operationally: concurrency and ISA. Flur, E. Gray, Pulte, Sarkar, Sezgin, Maranget, Deacon, Sewell. In POPL 2016.
An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. Gray, Kerneis, Mulligan, Pulte, Sarkar, Sewell. In MICRO 2015.

ISAs (Instruction-Set Architectures) in Sail

First: modest fragments of IBM Power, ARMv7, x86, to use with concurrency semantics

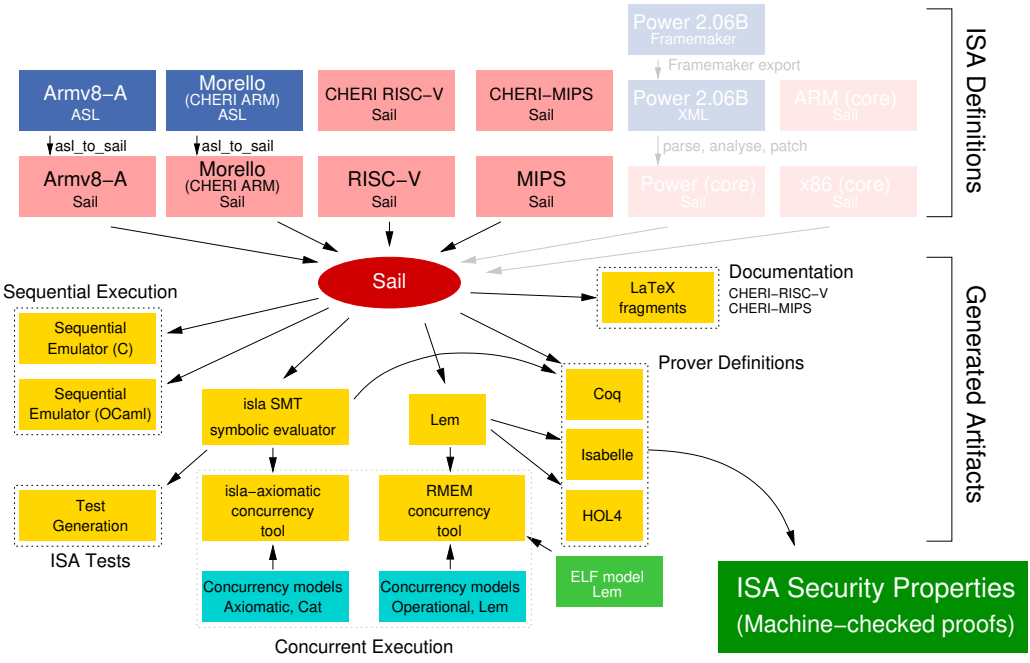
CHERI-MIPS, CHERI-RISC-V: We – the CHERI team – hand-wrote Sail specifications as part of the design process, and previously a CHERI-MIPS specification in Fox's L3

RISC-V: Historically only text. Our Sail specification is now adopted by RISC-V International

Armv8-A: Historically only pseudocode. Arm transitioned internally to mechanised ASL (Alasdair Reid et al.). We automatically translate that ASL to Sail

Morello: We automatically translate Arm-internal ASL to Sail

These last five are all complete sequential ISA models, e.g. able to boot an OS.


Power 2.06B
Framemaker
Framemaker export
Armv8–A
ASL
Morello
(CHERI ARM)
ASL
CHERI RISC–V
Sail
CHERI–MIPS
Sail
Power 2.06B
XML
ARM (core)
Sail
asl_to_sail
asl_to_sail
parse, analyse, patch
Armv8–A
Sail
Morello
(CHERI ARM)
Sail
RISC–V
Sail
MIPS
Sail
Power (core)
Sail
x86 (core)
Sail
ISA Definitions
Sail
Sequential Execution
LaTeX
fragments
Documentation
CHERI–RISC–V
CHERI–MIPS
Sequential
Emulator (C)
Sequential
Emulator (OCaml)
Prover Definitions
isla SMT
symbolic evaluator
Lem
Coq
Isabelle
HOL4
Test
Generation
isla–axiomatic
concurrency
tool
RMEM
concurrency
tool
ISA Tests
Concurrency models
Axiomatic, Cat
Concurrency models
Operational, Lem
ELF model
Lem
Concurrent Execution
Generated Artifacts
ISA Security Properties
(Machine–checked proofs)

Sail Armv8-A and Morello

Includes full ISA: Floating-point, address translation & page-table walks, synchronous exceptions, hypervisor mode, crypto instructions, vector instructions (NEON and SVE), memory partitioning and monitoring, pointer authentication, etc. . .

Such complete authoritative architecture descriptions not previously publicly available for formal reasoning

A sample Arm instruction, in the CompCert assembly abstraction

Consider `add x4, x3, #1`

CompCert 3.6 Armv8-A assembly semantics:

```
Inductive instruction: Type :=
| Paddimm (sz:isize) (rd:iregsp) (r1:iregsp) (n:Z) (**r addition*)
| ...

Definition exec_instr (f:function) (i:instruction) (rs:regset) (m:mem) :outcome :=
match i with
| Paddimm W rd r1 n ⇒ Next (nextinstr (rs*rd ← (Val.add rs*r1 (Vint (Int.repr n))))) m
| ...
```

A Sample Morello Instruction, in the full ASL/Sail semantics

LDR (literal), for loading a capability from a PCC-relative address

```
function clause __DecodeA64 ((pc, ([bitone,bitzero,bitzero,bitzero,bitzero,bitzero,
  bitone,bitzero,bitzero,bitzero,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_]
  as __opcode)) if SEE < 99) = {
    SEE = 99; let imm17 = Slice(__opcode, 5, 17); let Ct = Slice(__opcode, 0, 5);
    decode_LDR_C_I_C(imm17, Ct) }

val decode_LDR_C_I_C : (bits(17), bits(5)) -> unit
function decode_LDR_C_I_C (imm17, Ct) = {
    let 't = UInt(Ct);
    let offset : bits(64) = SignExtend(imm17 @ 0b0000, 64);
    execute_LDR_C_I_C(offset, t) }

val execute_LDR_C_I_C : forall ('t:Int),(0<='t & 't<=31). (bits(64),int('t)) -> unit
function execute_LDR_C_I_C (offset, t) = {
    CheckCapabilitiesEnabled();
    let base : VirtualAddress = VAFromCapability(PCC);
    let address : bits(64) = Align(VAddress(base) + offset, CAPABILITY_DBYTES);
    VACheckAddress(base, address, CAPABILITY_DBYTES, CAP_PERM_LOAD, AccType_NORMAL);
    data : bits(129) = MemC_read(address, AccType_NORMAL);
    let data : bits(129) = CapSquashPostLoadCap(data, base);
    C_set(t) = data }

val VACheckAddress : forall ('size : Int).
  (VirtualAddress, bits(64), int('size), bits(64), AccType) -> unit
function VACheckAddress (base, addr64, size, requested_perms, acctype) = {
    c : bits(129) = undefined;
    if VAIsBits64(base) then { c = DDC_read() }
    else { c = VAToCapability(base) };
    __ignore_15 = CheckCapability(c, addr64, size, requested_perms, acctype) }
```

```
val CheckCapability : forall ('size : Int).
  (bits(129), bits(64), int('size), bits(64), AccType) -> bits(64)
function CheckCapability (c, address, size, requested_perms, acctype) = {
    let el : bits(2) = AArch64_AccessUsesEL(acctype);
    let 'msbit = AddrTop(address, el);
    let s1_enabled : bool = AArch64_IsStageOneEnabled(acctype);
    addressforbounds : bits(64) = address; [...7 lines setting addressforbounds...]
    fault_type : Fault = Fault_None;
    if CapIsTagClear(c) then { fault_type = Fault_CapTag }
    else if CapIsSealed(c) then { fault_type = Fault_CapSeal }
    else if not_bool(CapCheckPermissions(c, requested_perms))
        then { fault_type = Fault_CapPerm }
    else if (requested_perms & CAP_PERM_EXECUTE) != CAP_PERM_NONE
            & not_bool(CapIsExecutePermitted(c)) then { fault_type = Fault_CapPerm }
    else if not_bool(CapIsRangeInBounds(c, addressforbounds, size[64 .. 0]))
        then { fault_type = Fault_CapBounds };
    if fault_type != Fault_None then {
        let is_store : bool = CapPermsInclude(requested_perms, CAP_PERM_STORE);
        let fault : FaultRecord = CapabilityFault(fault_type, acctype, is_store);
        AArch64_Abort(address, fault) };
    return(address) }
```

A Sample Morello Instruction, in the full ASL/Sail semantics

LDR (literal), for loading a capability from a PCC-relative address

call graph is 7 300 lines of Sail, including address translation

How can we use rigorous semantics in "lightweight formal" ways to ease conventional engineering in the CHERI development process?

"Lightweight Rigorous Engineering"

For CHERI-MIPS and CHERI-RISC-V

- use formal ISA semantics as central design artifacts (first in L3, then Sail)
  owned by CHERI systems and architecture researchers and engineers, not by semanticists

"Lightweight Rigorous Engineering"

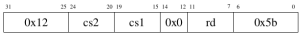
For CHERI-MIPS and CHERI-RISC-V

- use formal ISA semantics as central design artifacts (first in L3, then Sail)
  owned by CHERI systems and architecture researchers and engineers, not by semanticists
- embed in CHERI architecture specification documents

CToPtr

Format

CToPtr rd, cs1, cs2



Description

If the **tag** field of capability register *cs1* is not set then integer register *rd* is set to 0, otherwise integer register *rd* is set to *cs1*.**address** − *cs2*.**base**.

Semantics

```
let cs2_val = if unsigned(cs2) == 0 then DDC else C(cs2);
let cs1_val = C(cs1);
if not (cs2_val.tag) then {
  handle_cheri_reg_exception(CapEx_TagViolation, cs2);
  RETIRE_FAIL
} else if cs1_val.tag & isCapSealed(cs1_val) then {
  handle_cheri_reg_exception(CapEx_SealViolation, cs1);
  RETIRE_FAIL
} else {
  /* Note: returning zero for untagged values breaks magic constants such as SIG_IGN
     */
  X(rd) = if not (cs1_val.tag) then
            zeros()
          else
            cs1_val.address - getCapBaseBits(cs2_val);
  RETIRE_SUCCESS
```

"Lightweight Rigorous Engineering"

For CHERI-MIPS and CHERI-RISC-V

- use formal ISA semantics as central design artifacts (first in L3, then Sail)
  owned by CHERI systems and architecture researchers and engineers, not by semanticists
- embed in CHERI architecture specification documents
- make executable as a test oracle, auto-translating Sail/L3 to C/OCaml/SML
  ($\sim$ 400KIPS, booting FreeBSD in 4 min)
  - use for testing hardware against
  - use for software bring-up
- use for fast exploration of design alternatives (e.g. compression schemes)
- use for automatic test generation
- auto-translate to SMT and use to check properties

Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. Nienhuis, Joannou, Bauereiss, Fox, Roe, Campbell, Naylor, Norton, Moore, Neumann, Stark, Watson, Sewell. In Security and Privacy 2020.

"Lightweight Rigorous Engineering"

To make this work, we paid a lot of attention to the pragmatics:

- syntax like existing vendor pseudocode
- lightweight dependent type inference to check bitvector sizes
- expressive enough for full Arm-A/RISC-V, but otherwise as simple as possible
- auto-generate emulators fast enough to boot an OS
- integrated with concurrency semantics
  (not for CHERI, but important for RISC-V and Arm)

All this came first – then, when we got to mechanised proof about the specifications, they were well-established and well-validated.

"Lightweight Rigorous Engineering"

Our RISC-V formal ISA specification, written to support CHERI-RISC-V, has been adopted by RISC-V International

"Lightweight Rigorous Engineering"

For Morello

- Arm wrote ISA spec in their ASL, incorporating CHERI ideas into base Armv8.2-A ASL
- We auto-translate that ASL into Sail, working with weekly drops from Arm during Morello development
- We validate the result by cross-checking the Sail-generated C emulator and the Arm Morello Fast Model on the Arm Architecture Compliance Kit (ACK): 25 000+ tests covering Morello-specific functionality and base Armv8.2
- We generate interesting tests using Isla symbolic evaluation of the Sail model
- We measure specification coverage and adapt test generation accordingly
- Tests used in Arm as part of pre-silicon hardware and model testing
- Tests used in UCam for Morello QEMU bring-up

Verified security for the Morello capability-enhanced prototype Arm architecture.
Bauereiss, Campbell, T.Sewell, Armstrong, Esswood, Stark, Barnes, Watson, P. Sewell. ESOP 2022.

How can we establish confidence that the architecture *does* enforce the intended security protections?

What security properties do CHERI architectures aim to enforce?

Intuition: capabilities cannot be forged, so the available capabilities can't be increased during normal execution: *reachable capability monotonicity*

What security properties do CHERI architectures aim to enforce?

Intuition: capabilities cannot be forged, so the available capabilities can't be increased during normal execution: *reachable capability monotonicity*

Basis for fine-grain protection and secure encapsulation

What security properties do CHERI architectures aim to enforce?

Intuition: capabilities cannot be forged, so the available capabilities can't be increased during normal execution: *reachable capability monotonicity*

Basis for fine-grain protection and secure encapsulation

Make that (and more) mathematically precise as property of Sail/L3-generated Isabelle ISA specs, for Morello/CHERI-MIPS respectively

How do we establish confidence that it holds?

Whole-system property about machine execution of arbitrary code above the architecture

- conventional testing can never provide assurance
- the specs are big, so manual review can't either
  (62k/90k/210k lines of ASL/Sail/Isabelle for Morello, 6k lines L3 for CHERI-MIPS)

Instead, do mechanised proof, in Isabelle, that the property holds.

Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. Nienhuis, Joannou, Bauereiss, Fox, Roe, Campbell, Naylor, Norton, Moore, Neumann, Stark, Watson, Sewell. In Security and Privacy 2020.

Verified security for the Morello capability-enhanced prototype Arm architecture.
Bauereiss, Campbell, T.Sewell, Armstrong, Esswood, Stark, Barnes, Watson, P. Sewell. ESOP 2022.

How can we prove something about such a large spec?

Factor proof via properties of *instruction-local* semantics:

For capability-load, $[\![$`CLW rd,cb,offset`$]\!]$ is roughly the set of traces of shape:

$$[\text{ReadReg}(cb, c),\ \text{ReadMem}(addr, 4, c'),\ \text{WriteReg}(rd, c'')]$$

where c is an arbitrary capability value with load permission, with a virtual address and 4-byte footprint within its bounds; c' is an arbitrary value; and c'' is the appropriately sign-extended value.

...and many more traces for exceptional cases

How can we prove something about such a large spec?

Factor proof via properties of *instruction-local* semantics:

For capability-load, $[\![$`CLW rd,cb,offset`$]\!]$ is roughly the set of traces of shape:

$$[\text{ReadReg}(cb, c),\ \text{ReadMem}(addr, 4, c'),\ \text{WriteReg}(rd, c'')]$$

where c is an arbitrary capability value with load permission, with a virtual address and 4-byte footprint within its bounds; c' is an arbitrary value; and c'' is the appropriately sign-extended value.

...and many more traces for exceptional cases

(this is a tremendously useful abstraction when one starts working with real ISAs – also for concurrency semantics, program-logic reasoning, ...)

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

Property 1: Capability register writes

In any execution trace of a single instruction, for every write of a tagged capability to a register at a given point in the trace, one of the following holds:

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

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.

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.

Property 4: Memory accesses

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

...prove those hold across the entire ISA, then show they imply the top-level theorem

Reachable capabilities

The smallest set including:

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

Theorem (Reachable Capability Monotonicity)

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

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

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

Proof engineering

Combination of custom tactics within Isabelle and external generation of lemmas (and proof scripts)

37 000 generated lines
8 600 manually written lines
8 900 abstract model, monotonicity proof, proof tools
rely on SMT for part of proof about compression
proof executes in 3h23m in 18Gb

24 person-months proof
23 person-months preliminaries (CHERI abstraction, Sail-to-SMT, making model usable)
17 person-months test generation and ACK validation

Limitations

- properties of architecture, not hardware design
- w.r.t. the architectural abstraction, so no side-channel properties
- sequential only
- assume arbitrary fixed translation mapping (CHERI caps are wrt VAs)
- proof largely foundational in Isabelle, but rely on SMT for one lemma
- ASL-to-Sail not subject to verification (validated by testing)
- Sail-to-Isabelle (necessarily) trusted
- ASL subject to limitations documented by Arm wrt implementation-defined behaviour

Other ongoing CHERI and Architecture Semantics work

- Isla symbolic evaluation engine for Sail ISA semantics

  Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. Armstrong, Campbell, Simner, Pulte, Peter Sewell. In CAV 2021.

- architecture concurrency semantics for Armv8-A, RISC-V,... now including Armv8-A instruction-fetch and virtual memory

  Relaxed virtual memory in Armv8-A. Simner, Armstrong, Pichon-Pharabod, Pulte, Grisenthwaite, Sewell. In ESOP 2022.

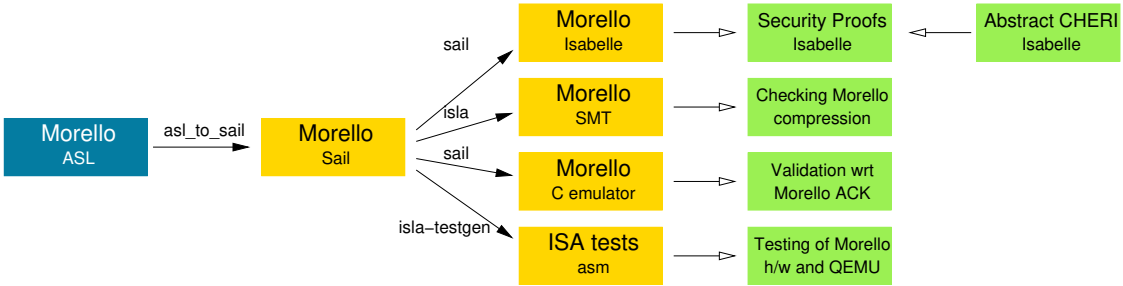
- relationships between CHERI C semantics and our work on C semantics, with ISO WG14 C standards committee

  Exploring C Semantics and Pointer Provenance. Memarian, Gomes, Davis, Kell, Richardson, Watson, Sewell. In POPL 2019

- integrating Isla symbolic evaluation with Iris program logic, for reasoning about specific Arm and RISC-V machine-code w.r.t. real ISA semantics

  Islaris: Verification of Machine Code Against Authoritative ISA Semantics. Sammler, Hammond, Lepigre, Campbell, Pichon-Pharabod, Dreyer, Garg, Sewell. Under submission.

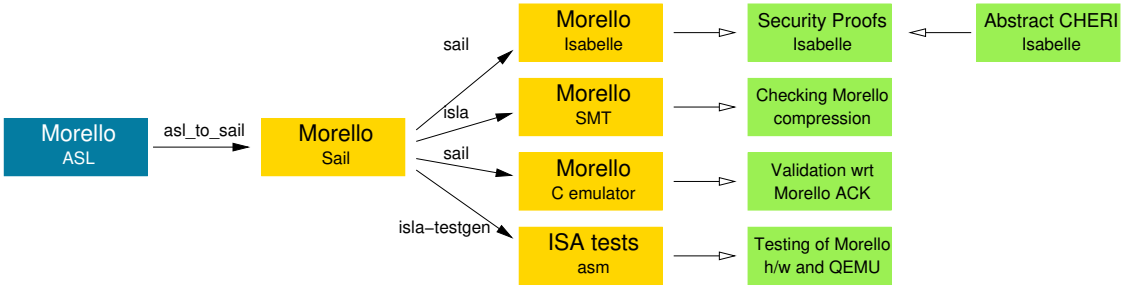
Morello ISA verification summary



- auto-translate that 62k LoS ASL into Sail (from weekly drops)
- use Sail to generate Isabelle (210k LoS) and SMT
- use Sail+SMT symbolic evaluation (Isla) to generate tests for h/w and QEMU
- use Isabelle for mechanised proof of general security properties

Found various security holes, including one not previously known, and other issues

Morello ISA verification summary



- auto-translate that 62k LoS ASL into Sail (from weekly drops)
- use Sail to generate Isabelle (210k LoS) and SMT
- use Sail+SMT symbolic evaluation (Isla) to generate tests for h/w and QEMU
- use Isabelle for mechanised proof of general security properties

Found various security holes, including one not previously known, and other issues

Machine-checked mathematical proofs of security properties of full-scale industry architecture