

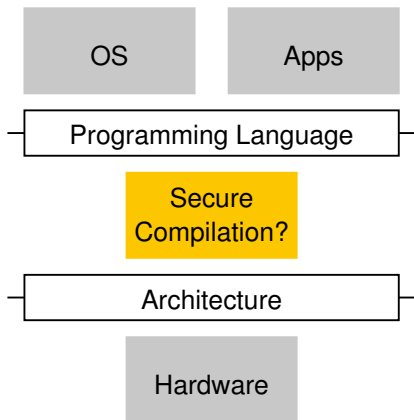
Secure Compilation – understanding the endpoints?

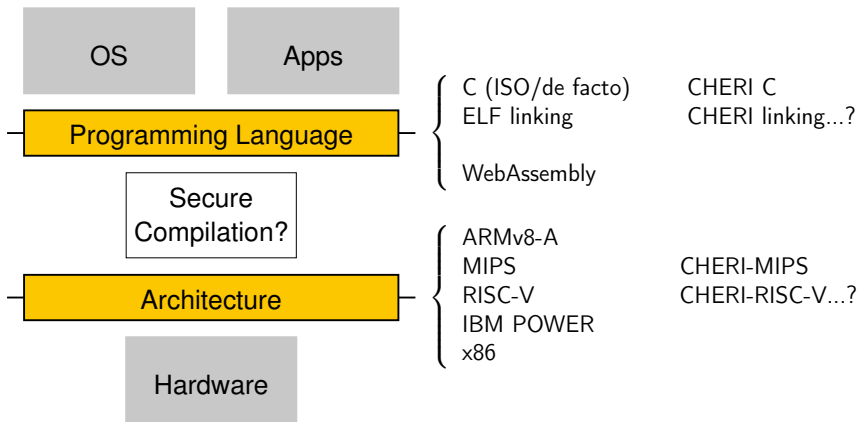
Peter Sewell

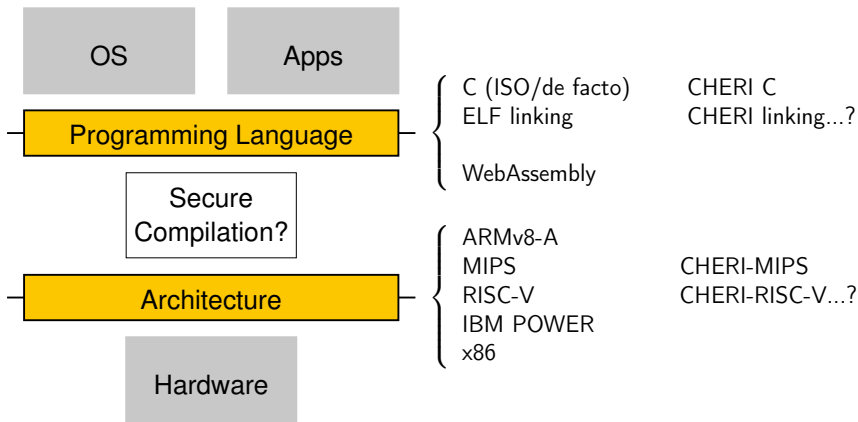
University of Cambridge

Secure Compilation: Dagstuhl Seminar 18201, May 13–18, 2018

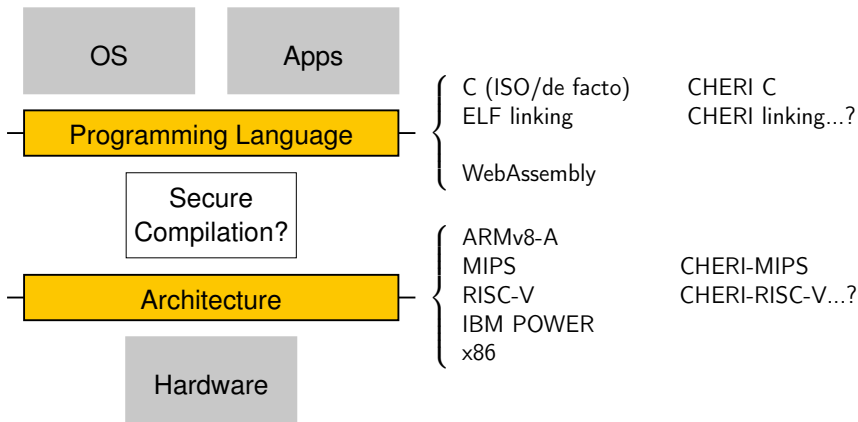
This work was partially supported by EPSRC grant EP/K008528/1 (REMS), an ARM iCASE award, and EPSRC IAA KTF funding. Approved for public release; distribution is unlimited. This research is sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237 ("CTSRD") and FA8650-18-C-7809 ("CIFV"). The views, opinions, and/or findings contained in this article/presentation are those of the author(s)/presenter(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.







Our focus: mainstream architectures and languages used to build operating systems – and CHERI – rather than clean-slate or higher-level languages



Aiming to

1. clarify what these abstractions *are* (for everybody), and
2. make re-usable semantic artifacts. For you!

Architecture Semantics

Architecture Semantics

Why is it hard, where did we get to, and what does this mean for secure compilation?

Architecture Semantics =

Assembly syntax ? (in progress)

+

Decode ✓

+

Instruction Semantics (sequential ISA behaviour) ✓ (FP,vec?)

+

User-mode Concurrency ✓

+

System Semantics (exceptions, interrupts, virtual memory,...) ?

+

System Concurrency ?

+

Debug architecture ??

Architecture Semantics =

Assembly syntax ? (in progress)

+

Decode ✓

+

Instruction Semantics (sequential ISA behaviour) ✓ (FP,vec?)

+

User-mode Concurrency ✓

+

System Semantics (exceptions, interrupts, virtual memory,...) ?

+

System Concurrency ?

+

Debug architecture ??

How complete do we need to be? For correctness of well-behaved code: sound fragment. For security properties of arbitrary code: complete model...

Instruction Semantics: Metalanguages

L3: Anthony Fox, and CHERI team for their model

- handwritten ISA models (ARM, x86, MIPS, RISC-V) for CakeML, and CHERI-MIPS (L3 model used as a primary design tool)

Sail (v1): Kathy Gray, Gabriel Kerneis, Christopher Pulte, Shaked Flur, Susmit Sarkar, Peter Sewell

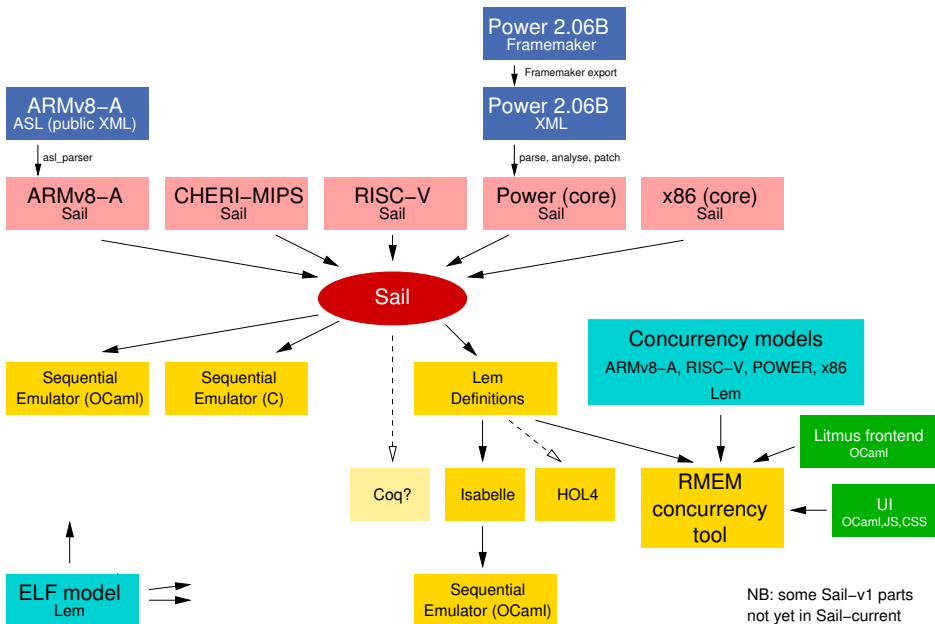
- various ISA models, handwritten and from IBM POWER XML, integrated with concurrency semantics

ASL (current): Alastair Reid and others at ARM

- ARMv8-A, ARMv8-M, and in-progress architecture extensions

Sail (current): Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Robert Norton-Wright, Mark Wassell, Neel Krishnaswami, Jon French, Prashanth Mundkur, Shaked Flur, Christopher Pulte, Ian Stark, Peter Sewell

- ARMv8-A, RISC-V, CHERI-MIPS, MIPS, ...



Challenge: legal

Challenge: legal

- 2008 no vendor provides machine-readable ISA description
 - Most don't have it, even internally (beyond “golden” C++ models)
 - NB: for ARM, this is part of their core IP

Challenge: legal

- 2008 no vendor provides machine-readable ISA description
 - Most don't have it, even internally (beyond “golden” C++ models)
 - NB: for ARM, this is part of their core IP
- 2008 we started talking with ARM about their concurrency architecture

Challenge: legal

- 2008 no vendor provides machine-readable ISA description
 - Most don't have it, even internally (beyond “golden” C++ models)
 - NB: for ARM, this is part of their core IP
- 2008 we started talking with ARM about their concurrency architecture
- 2011– Alastair Reid does heroic work to make ASL machine-readable

Challenge: legal

- 2008 no vendor provides machine-readable ISA description
 - Most don't have it, even internally (beyond “golden” C++ models)
 - NB: for ARM, this is part of their core IP
- 2008 we started talking with ARM about their concurrency architecture
- 2011– Alastair Reid does heroic work to make ASL machine-readable
- 2011–12 we started talking about ISA spec with Alastair and with ARM Lead Architect

Challenge: legal

2008 no vendor provides machine-readable ISA description
Most don't have it, even internally (beyond "golden" C++ models)
NB: for ARM, this is part of their core IP

2008 we started talking with ARM about their concurrency architecture

2011– Alastair Reid does heroic work to make ASL machine-readable

2011–12 we started talking about ISA spec with Alastair and with ARM
Lead Architect

2012–15 various meetings with ARM and ARM Legal

2015/9 UCam ASL LU licence in place

Challenge: legal

2008 no vendor provides machine-readable ISA description
Most don't have it, even internally (beyond “golden” C++ models)
NB: for ARM, this is part of their core IP

2008 we started talking with ARM about their concurrency architecture

2011– Alastair Reid does heroic work to make ASL machine-readable

2011–12 we started talking about ISA spec with Alastair and with ARM
Lead Architect

2012–15 various meetings with ARM and ARM Legal

2015/9 UCam ASL LU licence in place

2017/4 ARM release public v8.3-A machine-readable spec

Challenge: legal

2008 no vendor provides machine-readable ISA description
Most don't have it, even internally (beyond “golden” C++ models)
NB: for ARM, this is part of their core IP

2008 we started talking with ARM about their concurrency architecture

2011– Alastair Reid does heroic work to make ASL machine-readable

2011–12 we started talking about ISA spec with Alastair and with ARM
Lead Architect

2012–15 various meetings with ARM and ARM Legal

2015/9 UCam ASL LU licence in place

2017/4 ARM release public v8.3-A machine-readable spec

now we provide hopefully-usable Sail and Isabelle versions thereof

Challenge: readability

Keep close to engineer-friendly pseudocode; generate good LaTeX.

Sample CHERI instruction:

```
function clause execute (CIncOffsetImmediate(cd, cb, imm)) = {  
  checkCP2usable();  
  let cb_val = readCapReg(cb);  
  let imm64 : bits(64) = sign_extend(imm) in  
  if (register_inaccessible(cd)) then  
    raise_c2_exception(CapEx_AccessSystemRegsViolation, cd)  
  else if (register_inaccessible(cb)) then  
    raise_c2_exception(CapEx_AccessSystemRegsViolation, cb)  
  else if ((cb_val.tag) & (cb_val.sealed)) then  
    raise_c2_exception(CapEx_SealViolation, cb)  
  else  
    let (success, newCap) = incCapOffset(cb_val, imm64) in  
    if (success) then  
      writeCapReg(cd, newCap)  
    else  
      writeCapReg(cd, int_to_cap(to_bits(64, getCapBase(cb_val)) + imm64))  
}
```

Challenge: Scale

For small user-mode fragments of simple ISAs: trivial. RISC-V:

```
union clause ast = LOAD : (bits(12), regbits, regbits)
```

```
function clause decode
```

```
  (imm : bits(12)) @ (rs1 : regbits) @ 0b011 @ (rd : regbits) @ 0b0000011  
  = Some(LOAD(imm, rs1, rd))
```

```
function clause execute(LOAD(imm, rs1, rd)) =
```

```
  let addr : xlen_t = X(rs1) + EXT5(imm) in
```

```
  let result : xlen_t = MEMr(addr, 8) in
```

```
  X(rd) = result
```

For a production architecture, not so trivial. Full ARMv8.3-A 64-bit (including floating point, vector instructions, address translation, but not 32-bit): 20k LOS.

For ARM load-immediate instructions LDR <Xt>, [<Xn|SP>], #<simm>:
220 auxiliary functions (including address translation etc.)

Solution: automation is essential!

typechecking, generation $ASL \mapsto Sail \mapsto \{\text{emulators, prover definitions}\}$

Challenge: Lightweight Dependent Typing

The ARM specification involves bitvectors whose lengths are *computed* in complex ways. For example:

```
val FPThree : forall 'n, 'n in {16, 32, 64}. bits(1) -> bits('n)

function FPThree sign = {
  let 'E = (if 'n == 16 then 5 else if 'n == 32 then 8 else 11) : {|5, 8, 11|};
  let 'F = 'n - E - 1;
  exp = 0b1 @ Zeros(E - 1);
  frac = 0b1 @ Zeros(F - 1);
  return sign @ exp @ frac
}
```

This returns either a 16, 32, or 64-bit value, depending on the context within which it's called. The exponent is based on the length of the return type, with a type variable 'E for its size. We then create bitvectors involving 'E and the expected return length ('N), and the typechecker can check that they all are of the required length.

Solution: using modern type system design, all this can be *statically* checked, with lightweight dependent types, bidirectional typechecking, and Z3. Most types can be inferred.

Challenge: Complex Language Features

The ARM-internal ASL specification language has evolved over many years, accumulating tricky features: mutable local variables, recursive functions, exceptions and exception handlers, ...

To generate theorem-prover definitions, we have to translate these into simple mathematics.

Solution:

- ▶ keep Sail language as simple as possible – but no simpler
- ▶ use modern language-design tools for Sail itself (Ott, Menhir, OCaml); develop miniSail (in Isabelle and on paper)
- ▶ translate complex features away in back-ends, in multiple steps, preserving typability:
 - ▶ translation to monadic code
 - ▶ partial monomorphisation
 - ▶ rewriting of complex patterns (e.g. bitvector patterns)

Challenge: execution

Our primary uses all involve execution of models as test oracles:

Development of Concurrency Semantics (Sail v1):

- ▶ exhaustive concurrent execution: compute all allowed executions of concurrent litmus tests
- ▶ random concurrent execution: compute some allowed execution of concurrent algorithm

Supporting CHERI design and engineering work (L3 and Sail v2):

- ▶ sequential execution: use as emulator for software development (CHERI)
- ▶ sequential execution: use as reference for hardware development (CHERI)

Proof comes later...

Challenge: generating sequential emulators from model

Automatically generating a fast-enough emulator from the model provides a test oracle for hardware development and emulator for software – trivially updated as we update the architecture design.

Experience from CHERI-MIPS using SML emulator from L3 model: a few hundred kIPS is enough to bring up FreeBSD.

Solution: exploit Sail's type information to translate to C

Simple C benchmark: a loop incrementing a global, compiled GCC -O0, running in ARMv8-A AArch64 Sail model, without vector instructions, with address translation but not enabled:

Sail reference interpreter	4 IPS	
OCaml generation from Isabelle	100 IPS	
naive compilation to OCaml	7 000 IPS	
compilation to C, in progress	40 000 IPS	(RISC-V: more)

Challenge: integration with concurrency semantics

In relaxed architectures, inter-instruction semantics is non-SC.

- ▶ have to manage multiple in-flight instructions per thread – instruction-state continuations
- ▶ can (with care) still treat *intra*-instruction semantics sequentially
(so far, and excluding load-pair)
- ▶ but by no means *atomically*: have to expose register and memory accesses, the points at which addresses and values become determined, and dependencies.

Prompt Monad

```
type monad 'regval 'a 'e =
| Done of 'a
(* Read a number of bytes from memory, returned in little endian order *)
| Read_mem of read_kind * address * nat * (list memory_byte -> monad 'regval 'a 'e)
(* Tell the system a write is imminent, with given address and size *)
| Write_ea of write_kind * address * nat * monad 'regval 'a 'e
(* Request to write memory at last signalled address. Memory value should be 8
   times the size given in ea signal, given in little endian order *)
| Write_memv of list memory_byte * (bool -> monad 'regval 'a 'e)
(* Request a memory barrier *)
| Barrier of barrier_kind * monad 'regval 'a 'e
(* Read/write register requests *)
| Read_reg of register_name * ('regval -> monad 'regval 'a 'e)
| Write_reg of register_name * 'regval * monad 'regval 'a 'e
(* Request a Boolean value, chosen by the environment *)
| Undefined of (bool -> monad 'regval 'a 'e)
(* Print debugging or tracing information *)
| Print of string * monad 'regval 'a 'e
(* Result of a failed assert with possible error message to report *)
| Fail of string
| Exception of 'e
...
```

Challenge: making tools people can use

Solution: a whole lot of engineering. Sail→Lem→OCaml→ JavaScript;
ELF and Litmus frontends; DWARF; Test Libraries; JavaScript
GraphViz; GUI. rmem: <http://www.cl.cam.ac.uk/users/pes20/rmem>

Console - 100% + ↺ ↻ ×

Storage subsystem state (flat):
 Memory = [(0:4:0):W y/4=1, (1000:1:0):W x/4=0]

Thread 0 state: branch prediction targets: {}
 read issue order: []
 0:1 0x050000 **LDR W0,[X1]** mem reads: (0:1:0):R from (1000:1:0):W x/4=0 reg reads: R1=x from initialstate reg writes: R0=0x_63'0000000000000000
 0:2 0x050004 **EOR W2,W0,W0** reg reads: R0=0x_63'0000000000000000 from 0:2, R0=0x_63'0000000000000000 from 0:2 reg writes: R2=0x_63'0000000000000000 from 0:3 reg reads: R2=0x_63'0000000000000000
 2:0:3 **finish instruction**
 0:4 0x05000c **STR W2,[X3]** mem writes: (0:4:0):W y/4=1 reg reads: R3=y from initialstate, R2=0x_63'0000000000000001 from 0:4
 0:4 **complete store instruction**

Thread 1 state: branch prediction targets: {}
 read issue order: []
 1:1 0x051000 **LDR W0,[X1]** mem reads: (1:1:0):R from (1000:0:0):W y/4=0 reg reads: R1=y from initialstate
 1:1:1 **complete load instruction: 0**
 1:2 0x051004 **EOR W2,W0,W0**
 1:3 0x051008 **ADD W2,W2,#1** reg reads: R3=x from initialstate reg writes: R4=0x_63'0000000000000002
 1:4 0x051010 **MOV W4,#2** reg reads: R3=x from initialstate, R4=0x_63'0000000000000002 from 1:6
 1:6 0x051014 **STR W4,[X3]**
 3:1:6 **commit store instruction**

Choices so far (61):
 *0;1;2;2;3;3;4;5;0;0;0;0;1;1;1;1;2;3;4;5;2;3;1;1;0;0;0;0;0;0;2;2;2;2;2;2;2;2;0;0;3;0;5;3;3;3;3;3;4;4
 4 enabled transitions
 No disabled transitions

Step 62 (3/10 finished) Choose [0]:

Options: Execution Interface Graph Link to this state

Graph Refresh Download .dot centre - 100% + ↺ ↻ ×

Challenge: generation of prover definitions from model

Vital to get definitions into a clean idiomatic form – varies by prover.

For Isabelle and HOL4, one can use a bit-list representation, but really we have to translate away the bitvector length dependency, *monomorphising* the Sail specification with a Sail-to-Sail transform (duplicating and instantiating non-uniformly size-polymorphic functions), to use a bit-vector representation. We also use that for generating C emulator code.

For Coq, plan to keep dependency explicit.

For all provers there's a challenge to structure the definition in a way that lets the same definition be used for sequential and concurrent semantics. Either wrap prompt monad, or generate distinct versions.

Challenge: generation of prover definitions from model

RISC-V add instruction in Isabelle:

```
fun execute_ITYPE :: "12 word => 5 word => 5 word => iop => unit M " where
  " execute_ITYPE imm rs1 rd op1 = (
    (rX (regbits_to_regno rs1) :: ( 64 word) M) >=> (% rs1_val .
    (let (immext :: xlenbits) = ((EXTS ( 64 :: int) imm :: 64 word)) in
    (let (result :: xlenbits) = ((case op1 of
      RISCV_ADDI => (add_vec rs1_val immext :: 64 word)
      | ...)) in
    wX (regbits_to_regno rd) result))))"
```

FPZero from ARM in Isabelle:

```
definition FPZero :: "int => 1 word => (('N::len) word) M" where
  "FPZero (N :: int) sign = (
    (if (N = ( 16 :: int)) then
      ((let (F :: int) = 16 - 5 - 1 in
        (let (exp :: 5 bits) = Zeros__0 (make_the_value 5) in
        (let (frac :: 10 bits) = Zeros__0 (make_the_value F) in
        return (ucast (concat_vec (concat_vec sign exp :: 6 word) frac :: 'N word))))))
    else if (N = ( 32 :: int)) then ...
    else if (N = ( 64 :: int)) then ...
    else assert_exp False ((''(N == 16) || ((N == 32) || (N == 64))))'') >> exit ( ) )"
```

Monomorphisation has copied and instantiated function body for different bitvector lengths

Challenge: generation of prover definitions from model

RISC-V in hand-translated Coq:

Definition execute (ast : ast) : M unit :=

match ast **with**

```
| ITYPE (imm,rs1,rd,RISCV_ADDI) =>
  (rX (regbits_to_regno rs1) : M (mword 64)) >>= fun rs1_val =>
  let imm_ext : xlen_t := (EXTS (64:ii) imm : mword 64) in
  let result := (add_vec rs1_val imm_ext : mword 64) in
  wX (regbits_to_regno rd) result
...
```

Complex bitvector definitions in Coq need constraints (with automatic solving), type casts:

```
Definition ZeroExtend__0 {m} (x : bits m) N '{ArithFact (N >= 0)} : M (bits N) :=
  assert_exp' (Z.geb N m) "" >>= fun H =>
  returnm (cast_bits (concat_vec (Zeros__0 (N - m)) x)
    (ltac:(omega) : N - m + m = N) : bits N).
```

Challenge: validation

Sequential

- ▶ Random single-instruction testing (Pulte, for Sail v1 models)
- ▶ CHERI test suite (hand-crafted) (CHERI team)
- ▶ RISC-V test suite
- ▶ generation of test sequences from model (Campbell)
- ▶ booting OS – FreeBSD
- ▶ ARM conformance tests?

Concurrent

- ▶ litmus tests: accumulated library of 47169 (ARM, POWER, RISC-V, x86) Luc Maranget, Shaked Flur, Susmit Sarkar, Jade Alglave, Will Deacon, Linden Ralph, Peter Sewell..., both hand-written and generated using diy tool suite (Alglave, Maranget)

Regression

- ▶ Jenkins

Challenge: loose specification

Architects like to leave things loosely specified:

- ▶ particular register bit values
- ▶ behaviour for ambiguous page table entries
- ▶ choice of concurrent execution behaviour

Technical challenge for making models executable as test oracles

Annoying for testing h/w during development

Disaster for some security properties....

L3 CHERI-MIPS model determined

Sail ISA models: current state

Current Sail:

ARMv8.3-A 20 000 LOS **Armstrong, Bauereiss, Campbell, Reid, Norton-Wright, Sewell**
scope: all of public XML (sequential ISA, including FP, vectors, and address translation;
not system registers, interrupt controller, or debug arch)
generates: OCaml, C, and OCaml-from-Isabelle sequential emulation
generates: Isabelle (monomorphised, bitvector representation)

CHERI-MIPS 3900 LOS **Norton-Wright, CHERI-team – Joannou, Roe, Fox,...**
scope: everything (MIPS part: roughly MIPS 4k, no FP)
generates: OCaml and C sequential emulation
generates: Isabelle and HOL4 (monomorphised, bitvector representation)
validation: passes all CHERI test suite; boots FreeBSD

RISC-V 2700 LOS **Mundkur, Norton-Wright, Flur**
scope: sequential RV64-IMC (integer, multiplication, division, compressed instructions),
supervisor mode, most of machine mode
generates: OCaml sequential emulation
generates: Isabelle and HOL4 (bitvector representation)
validation: passes all supervisor and machine-mode tests

Sail v1:

Core user ISA fragments, integrated with rmem concurrency tool:

IBM POWER 5200 LOS (generated from IBM XML; **Kerneis, Gray**)

ARMv8-A 3900 LOS (**Flur**)

RISC-V 650 LOS (**Flur, Norton-Wright**)

x86 920 LOS (**Norton-Wright**, based on **Fox** L3 model)

Concurrency models: current state

User-mode concurrency

Operational models, including mixed-size, integrated with Sail (v1) ISA semantics:

ARMv8-A Christopher Pulte, Shaked Flur, Jon French, Susmit Sarkar, Luc Maranget, Peter Sewell – with Will Deacon at ARM

Co-developed with Deacon's ARM ARM axiomatic model; proved equivalent for common scope

RISC-V RISC-V Memory Model Task Group: Daniel Lustig and many others, including Shaked Flur, Christopher Pulte, Luc Maranget

Co-developed with axiomatic model; experimentally equivalent for common scope

IBM POWER Shaked Flur, Susmit Sarkar, Christopher Pulte, Luc Maranget, Peter Sewell – with Derek Williams at IBM

Early development co-informed with Alglave/Maranget axiomatic model development

x86 Shaked Flur, Linden Ralph, Christopher Pulte, Peter Sewell

Reasoning above these models? In its infancy...

Systems concurrency? Carefully ignored until now...

Architecture Semantics: the Ideal

A single specification, readable, mathematically rigorous, and executable as a test oracle, that suffices to:

- ▶ use as unambiguous and readable design documentation
- ▶ state and prove security properties of the architecture
- ▶ test hardware implementations directly against architecture
- ▶ automatically assess test-suite coverage of architecture
- ▶ generate good tests from the architecture
- ▶ model-check or do mechanised-proof-based-verification of hardware implementations with respect to the architecture
- ▶ test software above full range of architecturally-allowed behaviour
- ▶ build software model-checking or mechanised-proof-based verification tools above architectural assumption
- ▶ state and prove security properties of systems software idioms
- ▶ establish verified systems software, e.g. hypervisors (seL4), compilers (CompCert, CakeML), NSF DeepSpec, above vendor architectural assumption

Try it!

Sail and Sail ISA models in public github, BSD licence, OPAM:

<https://github.com/rem-s-project/sail>

generated Isa and HOL4: ARMv8.3-A (only Isa), CHERI-MIPS, RISC-V:

<https://github.com/rem-s-project/sail/tree/sail2/snapshots/>

Sail documentation (in progress):

<https://github.com/rem-s-project/sail/blob/sail2/manual.pdf>

Isabelle generation documentation:

<https://github.com/rem-s-project/sail/blob/sail2/snapshots/isabelle/Manual.pdf>

rmem concurrency tool – runs in your browser:

<http://www.cl.cam.ac.uk/users/pes20/rmem>

Alasdair Reid's blog post on ARMv8.3 XML release:

https://alastairreid.github.io/arm-v8_3/

Explicit speculation?

CHERI

CHERI

Architectural support for memory protection and compartmentalization

Cambridge and SRI, 2010–now

Hardware / Software / Formal co-design

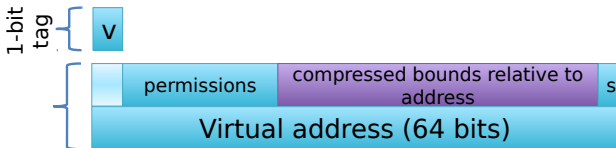
- ▶ CHERI-MIPS ISA formal models, Qemu-CHERI, FPGA prototypes
- ▶ CHERI abstract protection model, CHERI-MIPS concrete ISA
- ▶ CHERI Clang/LLVM, CheriBSD OS, C/C++-language apps

Robert N. M. Watson, Simon W. Moore, Peter G. Neumann
Jonathan Anderson, John Baldwin, Hadrien Barrel, Ruslan Bukin, David
Chisnall, Nirav Dave, Brooks Davis, Lawrence Esswood, Nathaniel W. Filardo,
Anthony Fox, Khilan Gudka, Alexandre Joannou, Robert Kovacsics, Ben
Laurie, A. Theo Markettos, J. Edward Maste, Alfredo Mazzinghi, Alan
Mujumdar, Prashanth Mundkur, Steven J. Murdoch, Edward Napierala,
Kyndylan Nienhuis, Robert Norton-Wright, Philip Paeps, Lucian Paul-Trifu,
Alex Richardson, Michael Roe, Colin Rothwell, Hassen Saidi, Peter Sewell,
Stacey Son, Domagoj Stolf, Andrew Turner, Munraj Vadera, Jonathan
Woodruff, Hongyan Xia, Bjoern A. Zeeb

CHERI

- ▶ Fine-grained protection (replacing all or some pointers with capabilities), and/or
- ▶ Secure encapsulation

Composes well with RISC archs, MMUs, C/C++, supporting incremental adoption. CHERI 128:



- ▶ Tags protect capabilities in registers and memory
Dereferencing an untagged capability throws an exception; in-memory overwrite automatically clears capability tag
- ▶ Bounds limit range of address space accessible via pointer
- ▶ Permissions limit operations – load, store, fetch
- ▶ Sealing for encapsulation: immutable, non-dereferenceable

CHERI Formal

L3 ISA model used as a main design tool since 2014

Alexandre Joannou, Mike Roe, Kyndylan Nienhuis, Brian Campbell,
plus Anthony Fox, Matthew Naylor for MIPS base

- using L3-generated emulator as reference for hardware implementation and for software bring-up

 - single-core BERI under MLton: 400 kIPS. < 5 min FreeBSD boot
 - multi-core CHERI: half that speed?

- HOL4 proofs about capability compression scheme (Fox)

- Isabelle proofs of security properties (Nienhuis)

Transitioning to Sail model, also to use in ISA spec (Norton-Wright and others)

Challenge: stating and proving architectural security properties

Kyndylan Nienhuis + CHERI team

From the CHERI architecture specification document:

Capability monotonicity *“is a property of the instruction set that any requested manipulation of a capability, whether in a capability register or in memory, either leads to strictly non-increasing rights, clearing of the capability tag, or a hardware exception. Monotonicity is preserved by capability-based instructions subject to their appropriate use (e.g., not requesting an increase in the length of a capability).”*

What do these mean? We have to make them mathematically precise – properties that are either true or false about the architecture specification.

Doing that often reveals that the original thinking was confused – e.g. we had several *different* notions of monotonicity in mind.

Plan for CHERI-MIPS

- ▶ Define and prove basic properties:
 - ▶ The guarantees of operations on capabilities (sealing, unsealing, calling, storing, loading and restricting)
 - ▶ The guarantees of operations on data (loading and storing)
- ▶ Define an abstraction of CHERI based on the eight operations mentioned above and prove that CHERI-MIPS implements it
- ▶ Define (in the abstraction) what the overall permissions of a compartment is and prove it monotonic
- ▶ Prove isolation of two-compartment reference-monitor example

Proofs based on Isabelle export from L3 CHERI-MIPS model

Current state: lots of lemmas; proofs of properties without CCALL.
Data operations still in progress

NB these are only some of the properties one would want

Operations on capabilities

- ▶ 22 instructions manipulate capabilities
- ▶ Types of manipulations:
 - ▶ Restricted and copied from reg to reg'
 - ▶ Loaded from $address$ to reg
 - ▶ Stored from reg to $address$
 - ▶ Sealed from reg to reg'
 - ▶ Unsealed from reg to reg'
 - ▶ Called reg and reg'
- ▶ For each type of manipulation we define which properties it guarantees

Guarantees of unsealing

From Isabelle

for all $s\ s'\ \text{auth}\ r\ r'$.

if $s' \in \text{NextStates}\ s$

and $\text{Sealed}\ \text{auth}\ r\ r' \in \text{DerivationsOfStep}\ s$

then let $t = \text{Cast}\ (\text{Base}\ (\text{CapReg}\ \text{auth}\ s)) +$
 $\text{Cast}\ (\text{Offset}\ (\text{CapReg}\ \text{auth}\ s))$ **in**

$\text{PermitSeal}\ (\text{CapReg}\ \text{auth}\ s)$

and $\text{Tag}\ (\text{CapReg}\ \text{auth}\ s)$

and not $\text{IsSealed}\ (\text{CapReg}\ \text{auth}\ s)$

and $\text{Cast}\ t \in \text{Segment}\ (\text{CapReg}\ \text{auth}\ s)$

and not $\text{IsSealed}\ (\text{CapReg}\ r\ s)$

and $\text{CapReg}\ r'\ s' = \text{CapReg}\ r\ s$ **with**

$\text{IsSealed} \leftarrow \text{True}, \text{ObjectType} \leftarrow t$

Theorem

The sealing guarantees hold in the L3 model of CHERI/MIPS.

Challenge: Proving Security Properties

- ▶ The CHERI-MIPS L3 model is large and under constant development
 - ▶ Complete enough to boot FreeBSD
 - ▶ 180 instructions
 - ▶ 10K non-comment, non-blank lines
 - ▶ 582 commits since 4 Apr 2014
- ▶ We developed a python script to generate parts of the proof
 - ▶ 11k manual non-comment, non-blank lines
 - ▶ 9k generated non-blank lines

Any change to the architecture spec might require corresponding updates to the proof – in principle perhaps large, but in fact we've been able to re-run proofs after significant changes without large updates: continuous-integration proof should be feasible.

Bugs found during proof

Proof aiming at *assurance* – but also finds bugs

Bugs found in CHERI-MIPS L3 architecture model:

- ▶ All capabilities allowed you to load data, even if they do not have the Load permission
- ▶ legacy store instructions always allowed you to store one byte past the authority of the capability in register 0
- ▶ overflow not correct in authority check, sometimes allowing stores to the end of the address range without permission
- ▶ ERET instruction should give unpredictable results when a capability branch was taken in the previous step, but the L3 model had a defined state

Easily fixable – but give evidence of discriminating nature of proof.

Several kinds:

- ▶ Misconceptions in human-readable spec – improperly thought out ideas.
- ▶ Miselaborations in human-readable spec – “stupid bugs” and “typos”
- ▶ Mistranscriptions from human-readable spec to L3/Sail – typically also “stupid bugs”

Also: better understanding of the problems with loosely specified instruction outcomes

Porting proof from L3-Isabelle to Sail-Isabelle?

- ▶ Determinise Sail model in same way as L3
- ▶ Make monads look the same – L3-Isabelle has a simple state monad; Sail-Isabelle currently has either state+nondet+exc monad or prompt monad.
- ▶ Handle contingent differences, e.g. in register state structure, and prove commutativity facts

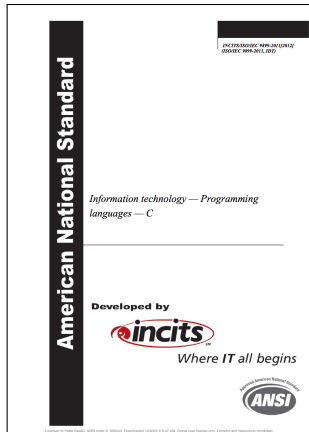
...presumably all doable, but not on our critical path right now

ISO C, de facto C, CHERI C, and Linker-speak

Source languages for Secure Compilation?

ISO C

WG14 committee:



“The Committee’s overall goal was to develop a clear, consistent, and unambiguous Standard for the C programming language which codifies the common, existing definition of C and which promotes the portability of user programs across C language environments.”

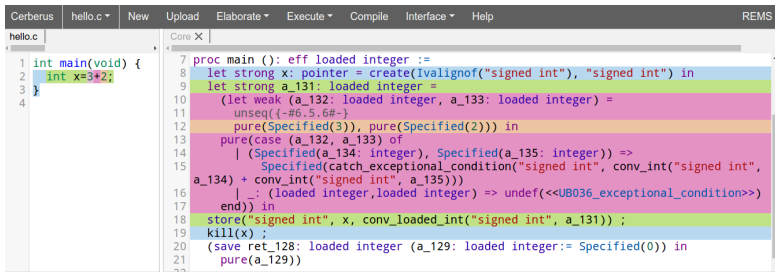
ISO C11 is 702 pages of prose. C++17 is 1622

Cerberus

Much of ISO *is* unambiguous (with sufficiently practiced reading...). How to capture that? Cerberus: elaboration to special-purpose Core (Kayvan Memarian, Victor Gomes, Peter Sewell)

```
int main(void) {  
    int x=3+2;  
}
```

looks simple – but already a lot going on in C: lifetime of *x*, unsequenced evaluation of arguments to *+*, integer conversions, UB-on-overflow



```
1 int main(void) {  
2   int x=3+2;  
3 }  
4
```

```
7 proc main (): eff loaded integer :=  
8   let strong x: pointer = create(Ialignof("signed int"), "signed int") in  
9   let strong a_131: loaded integer =  
10    (let weak (a_132: loaded integer, a_133: loaded integer) =  
11     unseq({-#6.5.6#-})  
12     pure(Specified(3)), pure(Specified(2))) in  
13     pure(case (a_132, a_133) of  
14      | (Specified(a_134: integer), Specified(a_135: integer)) =>  
15       Specified(catch_exceptional_condition("signed int", conv_int("signed int",  
16        a_134) + conv_int("signed int", a_135)))  
17      | _: (loaded integer, loaded integer) => undef(<<UB036_exceptional_condition>>))  
18      end)) in  
19     store("signed int", x, conv_loaded_int("signed int", a_131)) ;  
20     kill(x) ;  
21     (save ret_128: loaded integer (a_129: loaded integer:= Specified(0)) in  
22      pure(a_129))
```

Cerberus

Much of ISO *is* unambiguous (with sufficiently practiced reading...).
How to capture that? Cerberus: elaboration into special-purpose Core

But the *memory object semantics* is less so, and/or differs with
de-facto C practice: pointer provenance, uninitialised values, etc.

...many of the same issues that come up for CHERI C and in porting
system software to it (David Chisnall and rest of CHERI team)

Cerberus

Much of ISO *is* unambiguous (with sufficiently practiced reading...).
How to capture that? Cerberus: elaboration into special-purpose Core

But the *memory object semantics* is less so, and/or differs with
de-facto C practice: pointer provenance, uninitialised values, etc.

...many of the same issues that come up for CHERI C and in porting
system software to it (David Chisnall and rest of CHERI team)

Trying to understand what these are, what de facto practice is, and
to engage with WG14 to improve things for C2x

Try it! <http://www.cl.cam.ac.uk/~pes20/cerberus/>

What's a pointer value?

A simple numeric address?

What's a pointer value?

A simple numeric address?

At runtime, normally yes. But WG14 DR260 CR (2001):

“Implementations are permitted to track the origins of a bit-pattern [...] They may also treat pointers based on different origins as distinct even though they are bitwise identical.”

What's a pointer value?

A simple numeric address?

At runtime, normally yes. But WG14 DR260 CR (2001):

“Implementations are permitted to track the origins of a bit-pattern [...] They may also treat pointers based on different origins as distinct even though they are bitwise identical.”

Exploited in practice by some compilers: alias analysis uses provenance to reason that two pointers are distinct (even if they might have the same runtime numeric address), with optimisations assuming that is correct (relying on UB otherwise)

What's a pointer value?

A simple numeric address?

At runtime, normally yes. But WG14 DR260 CR (2001):

"Implementations are permitted to track the origins of a bit-pattern [...] They may also treat pointers based on different origins as distinct even though they are bitwise identical."

Exploited in practice by some compilers: alias analysis uses provenance to reason that two pointers are distinct (even if they might have the same runtime numeric address), with optimisations assuming that is correct (relying on UB otherwise)

But what does DR260CR really mean? It was never incorporated into the standard, and there are lots of choices – which determine what code is legal, and what alias analysis & optimisation is allowed to do.

Q1'. Must the pointer used for a memory access have the right provenance, i.e. be derived from the pointer to the original allocation (UB otherwise)? **Yes**

Example provenance_basic_global_xy.c:

```
int x=1, y=2;
...
int *p = &x + 1;
int *q = &y;
if (memcmp(&p, &q, sizeof(p)) == 0) {
    *p = 11; // does this have undefined behaviour?
    printf("x=%d y=%d *p=%d *q=%d\n", x, y, *p, *q);
}
```

(NB these are tests of tricky edge-cases – not necessarily sensible code!)

p has the provenance of the x allocation, but is used to access memory outside that, so the access gives UB

Provenance: Proposed semantics

Associate a *provenance* to every pointer and integer value:

- ▶ a single provenance ID, freshly chosen at each allocation
- ▶ the “empty” provenance

(in the C abstract machine, not at runtime in normal impls!)

Check provenance on all accesses, with UB otherwise:

- ▶ access via a pointer value with a single provenance ID must be within the memory footprint of the corresponding original allocation
- ▶ access via a pointer value with empty provenance is undefined behaviour (except device memory)

Then take care which operations preserve provenance, and look at the tricky cases: forming pointers by arithmetic, representation manipulation, and IO...

Making progress?

Closely related problems:

- ▶ C source semantics – ISO and de facto
- ▶ C source semantics – CHERI C
- ▶ C source semantics – “well-behaved” C
- ▶ LLVM and other compiler-intermediate-language semantics
- ▶ Rust unsafe-region semantics

ISO WG14 Memory Object Model working group

Discussion of provenance, uninitialised reads, padding, etc.

Meet later this week?

Questions and examples: *Clarifying Pointer Provenance v4*

<http://www.cl.cam.ac.uk/~pes20/cerberus/>

Composition?

It's not all about the ML module system...

Composition?

- ▶ inter-module binding (matching definitions with uses)

Composition?

- ▶ inter-module binding (matching definitions with uses)
- ▶ memory placement (placing section X at address Y)
- ▶ memory layout (placing sections relative to each other)
 - ▶ e.g. C compiler can't guarantee that global objects are placed in order of declaration – but need that for page tables
 - ▶ solution: put these in different sections using attributes
 - ▶ ... then use the linker script to order them.

Composition?

- ▶ inter-module binding (matching definitions with uses)
- ▶ memory placement (placing section X at address Y)
- ▶ memory layout (placing sections relative to each other)
 - ▶ e.g. C compiler can't guarantee that global objects are placed in order of declaration – but need that for page tables
 - ▶ solution: put these in different sections using attributes
 - ▶ ... then use the linker script to order them.
- ▶ inter-module encapsulation (local vs global symbols; but also hidden, protected, ...)
- ▶ inter-module versioning (symbol versions)
- ▶ inter-module binding tweaks (interposable or no? -Bsymbolic)
- ▶ link-time deduplication ('comdat', e.g. for OOL copies of inline function)
- ▶ build-time flexibility & configuration (weak alias idioms)
- ▶ extensibility (weak undefined symbols)
- ▶ instrumentation (-wrap, LD_PRELOAD)
- ▶ introspection (begin, end & etext symbols, run-time access to symbols or dynamic linker metadata, ...)
- ▶ ...

All this “linker-speak” is really part of the *language*.

Stephen Kell, Dominic Mulligan, Peter Sewell [OOPSLA 2016]

<https://github.com/rem-s-project/linksem>

Partial spec of ELF, DWARF, and static linking



Interesting source and target for secure compilation?

WebAssembly semantics (Conrad Watt)

- ▶ WebAssembly: a new universal bytecode for the web, supported by all browsers. [Haas et al., PLDI 2017]
- ▶ Full mechanised formal semantics, with type soundness proofs [Conrad Watt, CPP 2018 and Archive of Formal Proofs www.cl.cam.ac.uk/users/caw77/papers/mechanising-and-verifying-the-webassembly-specification.pdf and <http://isa-afp.org/entries/WebAssembly.html>]
- ▶ Verified interpreter and typechecker.
- ▶ Already discovered several bugs in specification [Rossberg, <https://github.com/WebAssembly/spec/commit/772d87705ea4786c0d44d41902097e91cf31f82b>]
- ▶ Ongoing work on memory model for the threads proposal.
- ▶ Ongoing work on constant-time WASM with Stefan et al., UCSC^{48/48}