Fixing the Foundations with Semantics and Capabilities: ARM, RISC-V, and CHERI

Peter Sewell

University of Cambridge

VeTTS workshop, Cambridge, 2019-09-24

This work was partially supported by EPSRC grant EP/K008528/1 (REMS), ERC AdG 789108 ELVER, ARM iCASE awards, 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.

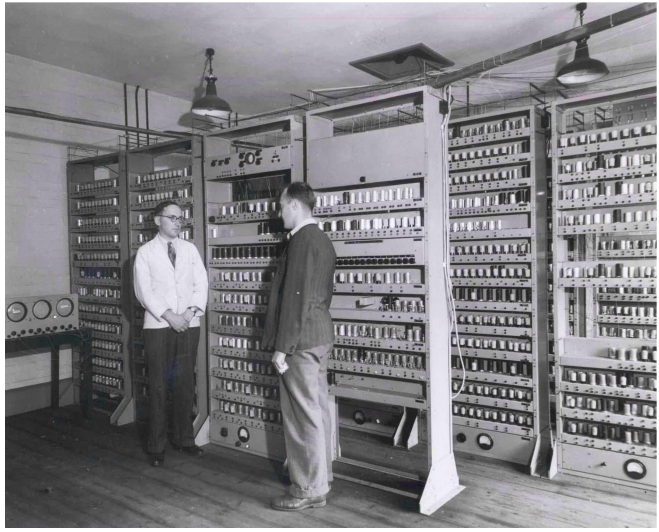
ISA semantics for ARM and RISC-V

Foundation for software verification: **architecture**

- the hardware/software interface, defining the envelope of all allowed hardware behaviour

(distinct from *microarchitecture* – hardware implementations)

Architecture $\simeq$ Instruction-set Architecture (ISA) + Concurrency



EDSAC, 1947–1958

EDSAC Order Code

Order	Explanation
$A\ n$	Add the number in storage location n into the accumulator.
$S\ n$	Subtract the number in storage location n from the accumulator.
$H\ n$	Transfer the number in storage location n into the multiplier register.
$V\ n$	Multiply the number in storage location n by the number in the multiplier register and add into the accumulator.
$N\ n$	Multiply the number in storage location n by the number in the multiplier register and subtract from the contents of the accumulator.
$T\ n$	Transfer the contents of the accumulator to storage location n, and clear the accumulator.
$U\ n$	Transfer the contents of the accumulator to storage location n, and do not clear the accumulator.
$C\ n$	Collate the number in storage location n with the number in the multiplier register, i.e., add a 1 into the accumulator in digital positions where both numbers have a 1 and a 0 in other digital positions.
$R\ 2^{n-2}$	Shift the number in the accumulator n places to the right, i.e., multiply it by 2^{-n}.
$L\ 2^{n-2}$	Shift the number in the accumulator n places to the left, i.e., multiply it by 2^{n}.
$E\ n$	If the number in the accumulator is greater than or equal to zero, execute next the order which stands in storage location n; otherwise, proceed serially.
$G\ n$	If the number in the accumulator is less than zero, execute next the order which stands in storage location n; otherwise, proceed serially.
$I\ n$	Read the next row of holes on the tape, and place the resulting 5 digits in the least significant places of storage location n.
$O\ n$	Print the character now set up on the teleprinter, and set up on the teleprinter the character represented by the five most significant digits in storage location n.
$F\ n$	Place the five digits which represent the character next to be printed by the teleprinter in the five most significant places of storage location n.
Y	Round off the number in the accumulator to 34 binary digits.
Z	Stop the machine, and ring the warning bell.

EDSAC Order Code

Order	Explanation
$A\ n$	Add the number in storage location n into the accumulator.
$S\ n$	Subtract the number in storage location n from the accumulator.
$H\ n$	Transfer the number in storage location n into the multiplier register.
$V\ n$	Multiply the number in storage location n by the number in the multiplier register and add into the accumulator.
$N\ n$	Multiply the number in storage location n by the number in the multiplier register and subtract from the contents of the accumulator.
$T\ n$	Transfer the contents of the accumulator to storage location n, and clear the accumulator.
$U\ n$	Transfer the contents of the accumulator to storage location n, and do not clear the accumulator.
$C\ n$	Collate the number in storage location n with the number in the multiplier register, i.e., add a 1 into the accumulator in digital positions where both numbers have a 1 and a 0 in other digital positions.
$R\ 2^{n-2}$	Shift the number in the accumulator n places to the right, i.e., multiply it by 2^{-n}.
$L\ 2^{n-2}$	Shift the number in the accumulator n places to the left, i.e., multiply it by 2^{n}.

Y	Round off the number in the accumulator to 34 binary digits.
Z	Stop the machine, and ring the warning bell.

One "Simple" ARM instruction

```
ADD <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}
```

"Add (immediate) adds a register value and an optionally-shifted immediate value, and writes the result to the destination register."

One "Simple" ARM instruction

`ADD <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}`

"Add (immediate) adds a register value and an optionally-shifted immediate value, and writes the result to the destination register."

CompCert ARMv8-A Semantics

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

(** Execution of a single instruction [i] in initial state [rs] and [m].  Return
updated state. For instructions that correspond to actual AArch64 instructions,
the cases are straightforward transliterations of the informal descriptions
given in the ARMv8 reference manuals.  *)

Definition exec_instr (f:function) (i:instruction) (rs:regset) (m:mem) : outcome :=
| Paddimm W rd r1 n =>
   Next (nextinstr (rs#rd <- (Val.add rs#r1 (Vint (Int.repr n))))) m
| ...
```

So why is this a big deal?

- scale
  - instructions are not really that simple
    (average 64-bit ARM instruction have 800 calls to auxiliary functions)

Decoding, 1/2

```
val decode64 : bits(32) -> unit
  effect {configuration, escape, undef, wreg, rreg, rmem, wmem}

function clause decode64
  ((_:bits(1) @ 0b0010001 @ _:bits(24) as op_code) if SEE<1066) = {
    SEE = 1066;
    Rd : bits(5) = op_code[4 .. 0];
    Rn : bits(5) = op_code[9 .. 5];
    imm12 : bits(12) = op_code[21 .. 10];
    shift : bits(2) = op_code[23 .. 22];
    S : bits(1) = [op_code[29]];
    op : bits(1) = [op_code[30]];
    sf : bits(1) = [op_code[31]];
    addsub_immediate_decode(Rd, Rn, imm12, shift, S, op, sf)
}
```

Decoding, 2/2

```
val addsub_immediate_decode :
    (bits(5), bits(5), bits(12), bits(2), bits(1), bits(1), bits(1))
    -> unit
    effect {escape, rreg, undef, wreg}

function addsub_immediate_decode(Rd, Rn, imm12, shift, S, op, sf) = {
    __unconditional = true;
    let 'd = UInt(Rd);  let 'n = UInt(Rn);
    let 'datasize = if sf == 0b1 then 64 else 32;
    let sub_op = op == 0b1;  let setflags = S == 0b1;
    imm : bits('datasize) = undefined : bits('datasize);
    match shift {
      0b00 => { imm = ZeroExtend(imm12, datasize) },
      0b01 => { imm = ZeroExtend(imm12 @ Zeros(12), datasize) },
      0b10 => { throw(Error_See("ADDG, SUBG")) },
      0b11 => { ReservedValue() }
    };
    __PostDecode();
    addsub_immediate(d, datasize, imm, n, setflags, sub_op)
}
```

Execution

```
function addsub_immediate(d, datasize, imm, n, setflags, sub_op) = {
    result : bits('datasize) = undefined : bits('datasize);
    let operand1 : bits('datasize) = if n == 31 then SP() else X(n);
    operand2 : bits('datasize) = imm;
    nzcv : bits(4) = undefined : bits(4);
    carry_in : bits(1) = undefined : bits(1);
    if sub_op then {
        operand2 = ~(operand2);
        carry_in = 0b1
    } else {
        carry_in = 0b0
    };
    (result, nzcv) = AddWithCarry(operand1, operand2, carry_in);
    if setflags then {
        (PSTATE.N @ PSTATE.Z @ PSTATE.C @ PSTATE.V) = nzcv
    };
    if d == 31 & ~(setflags) then { SP() = result }
    else { X(d) = result }
}
```

Auxiliary register-access functions

Register getters and setters defined as functions, e.g. those `SP() = result` and `X(n)`

```
function aset_SP(value) = {
    assert('width == 32 | 'width == 64);
    if PSTATE.SP == 0b0 then {
        SP_EL0 = ZeroExtend(value)
    } else {
        match PSTATE.EL {
          el if el == EL0 => SP_EL0 = ZeroExtend(value),
          el if el == EL1 => SP_EL1 = ZeroExtend(value),
          el if el == EL2 => SP_EL2 = ZeroExtend(value),
          el if el == EL3 => SP_EL3 = ZeroExtend(value)
        }
    }
}
val aget_X : forall 'width 'n, 0 <= 'n <= 31 & 'width in {8, 16, 32, 64}).
    (implicit('width), int('n)) -> bits('width) effect {rreg}

function aget_X(width, n) =
    if n != 31 then slice(_R[n], 0, width) else Zeros(width)
```

Execution auxiliary functions

```
val AddWithCarry : forall ('N : Int), ('N >= 0 & 'N >= 0).
  (bits('N), bits('N), bits(1)) -> (bits('N), bits(4))

function AddWithCarry (x, y, carry_in) = {
    let 'unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    let 'signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
    let result : bits('N) = __GetSlice_int('N, unsigned_sum, 0);
    let n : bits(1) = [result['N - 1]];
    let z : bits(1) = if IsZero(result) then 0b1 else 0b0;
    let c : bits(1) = if UInt(result) == unsigned_sum then 0b0 else 0b1;
    let v : bits(1) = if SInt(result) == signed_sum then 0b0 else 0b1;
    return((result, ((n @ z) @ c) @ v))
}
```

So why is this a big deal?

- scale
  - instructions are not really that simple
    (average 64-bit ARM instruction have 800 calls to auxiliary functions)
  - instruction sets are not that small
    (the ARMv8.4-A manual is 7476 pages)
- full-scale definitive machine-readable semantics did not exist for any major architecture, even within vendors (until Reid for ARM)
- legal concerns with making them publicly available
- readability for practising engineers
- usability as an executable test oracle
- usability for proof
- integration with concurrency semantics

Architecture ISA semantics, in Sail

Sail: a clean engineer-friendly first-order imperative language with lightweight dependent types (typechecked using SMT) for ISA specification, that can generate executable emulators and prover definitions.

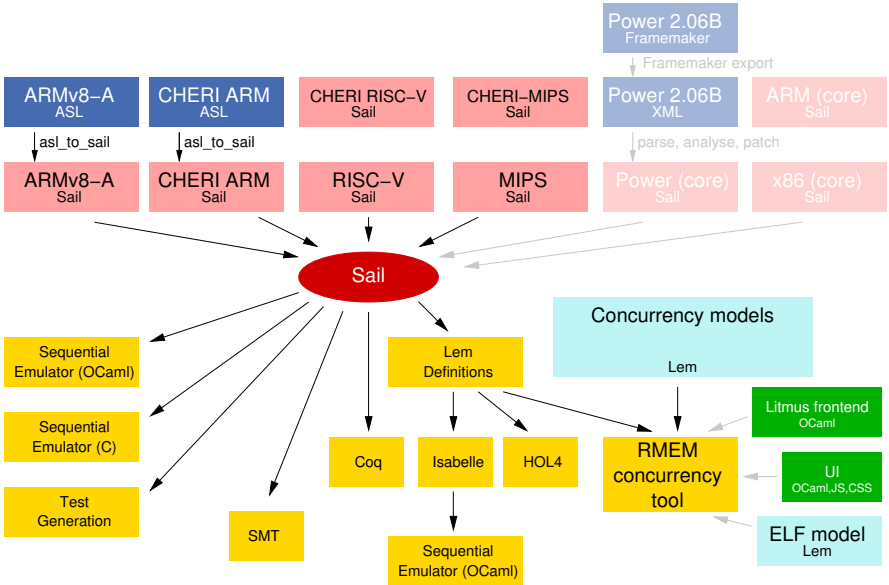
Legal and technical work with ARM to take their internal definition, make it available in Sail, and validate it.

Sail definitions can include:

- instruction execution: functions above register/memory primitives
- instruction AST
- assembly syntax and decode: bidirectional mappings

[Armstrong, Bauereiss, Campbell, Reid, Gray, Norton, Mundkur, Wassell, French, Pulte, Flur, Stark, Krishnaswami, Sewell; POPL 2019]

Sail Tooling



Sail ISA Models

	source	KLoS	KIPS	provers	boots	conc
ARMv8.5-A	ASL	125	200	Isa, HOL4, Coq*	Linux, Hafnium	
MIPS	hand	2	800	Isa, HOL4, Coq	FreeBSD	
RISC-V	hand	5		Isa, HOL4, Coq	Linux, FreeBSD	RMEM

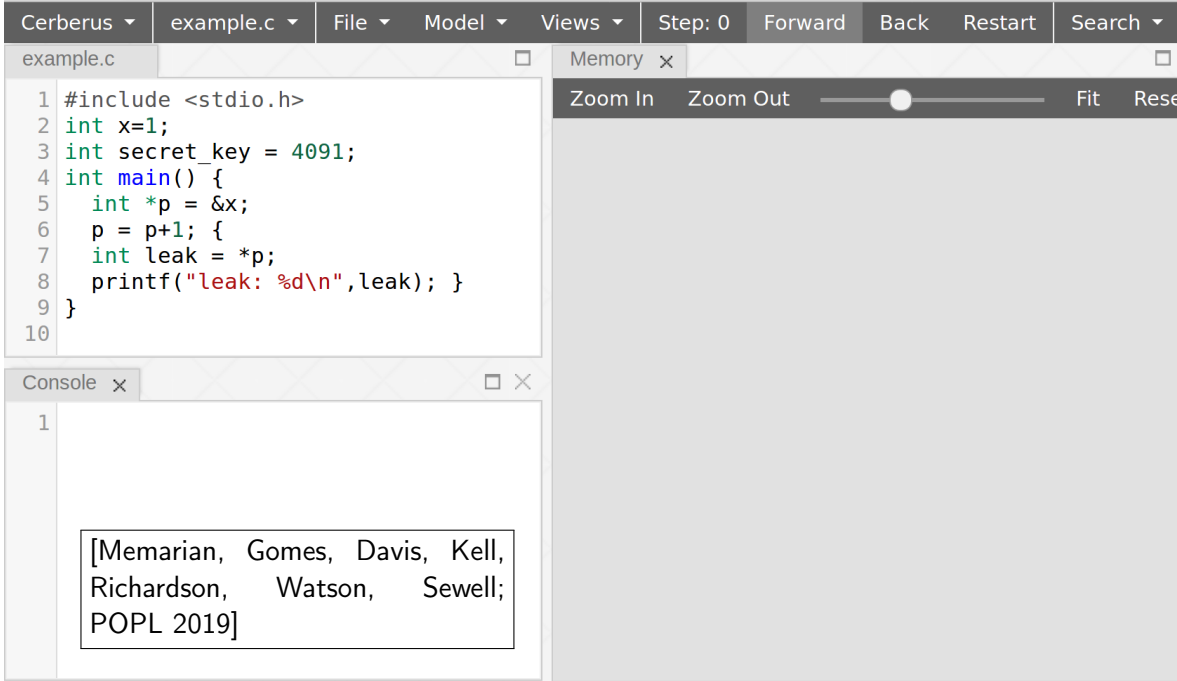
...and smaller IBM POWER and x86 fragments

All publicly available on github, `https://www.cl.cam.ac.uk/~pes20/sail/`.

BSD or BSD Clear licences

Viable for proof? Isabelle and (ongoing) Coq proofs about ARMv8.5-A address translation

Why CHERI? See C...


Cerberus
example.c
File
Model
Views
Step: 0
Forward
Back
Restart
Search
example.c
1 #include <stdio.h>
2 int x=1;
3 int secret_key = 4091;
4 int main() {
5 int *p = &x;
6 p = p+1; {
7 int leak = *p;
8 printf("leak: %d\n",leak); }
9 }
10
Console
1
[Memarian, Gomes, Davis, Kell, Richardson, Watson, Sewell; POPL 2019]
Memory
Zoom In
Zoom Out
Fit
Rese

Cerberus | example.c | File | Model | Views | Step: 8 | Forward | Back | Restart | Search

example.c

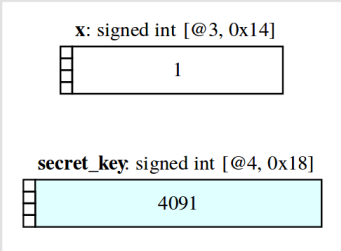
```
1  #include <stdio.h>
2  int x=1;
3  int secret_key = 4091;
4  int main() {
5    int *p = &x;
6    p = p+1; {
7    int leak = *p;
8    printf("leak: %d\n",leak); }
9  }
10
```

Console

```
1
```

Memory

Zoom In | Zoom Out | Fit | Rese



Cerberus | example.c | File | Model | Views | Step: 11 | Forward | Back | Restart | Search

example.c

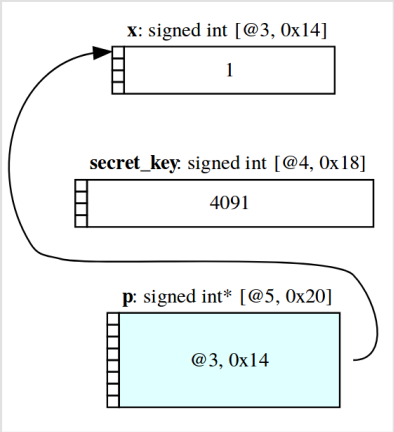
```
1  #include <stdio.h>
2  int x=1;
3  int secret_key = 4091;
4  int main() {
5    int *p = &x;
6    p = p+1; {
7    int leak = *p;
8    printf("leak: %d\n",leak); }
9  }
10
```

Console

```
1
```

Memory

Zoom In | Zoom Out | Fit | Rese



Cerberus | example.c | File | Model | Views | Step: 12 | Forward | Back | Restart | Search

example.c

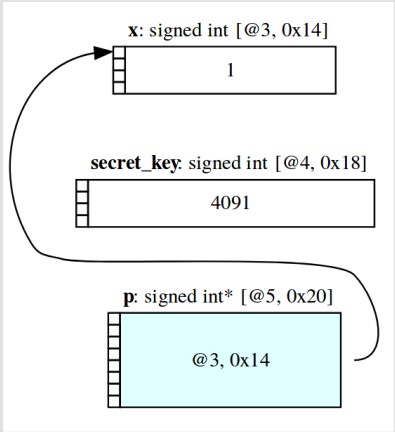
```
1  #include <stdio.h>
2  int x=1;
3  int secret_key = 4091;
4  int main() {
5    int *p = &x;
6    p = p+1; {
7    int leak = *p;
8    printf("leak: %d\n",leak); }
9  }
10
```

Console

```
1
```

Memory

Zoom In | Zoom Out | Fit | Rese



Cerberus | example.c | File | Model | Views | Step: 15 | Forward | Back | Restart | Search

example.c

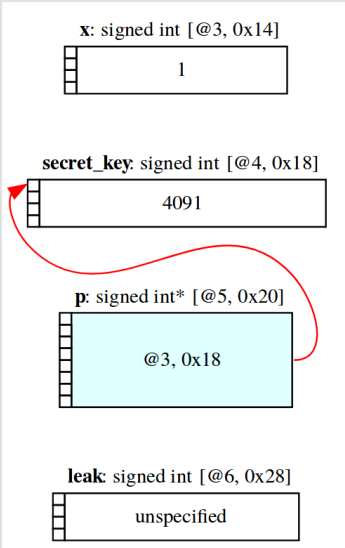
```
1  #include <stdio.h>
2  int x=1;
3  int secret_key = 4091;
4  int main() {
5    int *p = &x;
6    p = p+1; {
7    int leak = *p;
8    printf("leak: %d\n",leak); }
9  }
10
```

Console

```
1
```

Memory

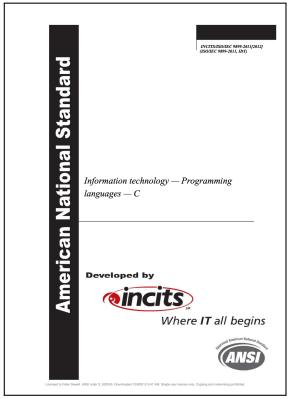
Zoom In | Zoom Out | Fit | Rese



```
$ gcc -std=c11 -O0 -Wall -pedantic 35c3.c
$ ./a.out
leak: 4091
```

Does C really permit that?

Yes and No...



- ISO C Standard (WG14) says `35c3.c` has *undefined behaviour (UB*
- ...it doesn't constrain C implementations for programs with UB
- I.e., it's the programmer's responsibility to avoid UB
- ...and compilers can assume its absence

Cerberus | example.c | File | Model | Views | Step: 15 | Forward | Back | Restart | Search

example.c

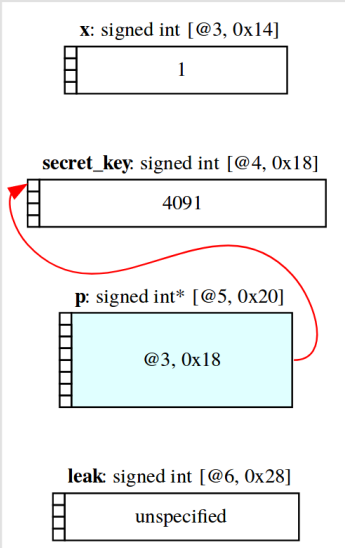
```
1  #include <stdio.h>
2  int x=1;
3  int secret_key = 4091;
4  int main() {
5    int *p = &x;
6    p = p+1; {
7    int leak = *p;
8    printf("leak: %d\n",leak); }
9  }
10
```

Console

```
1
```

Memory

Zoom In | Zoom Out | Fit | Rese



Cerberus | example.c | File | Model | Views | Step: 16 | Forward | Back | Restart | Search

example.c

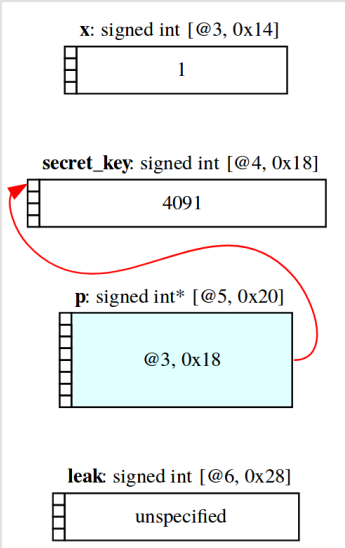
```
1  #include <stdio.h>
2  int x=1;
3  int secret_key = 4091;
4  int main() {
5    int *p = &x;
6    p = p+1; {
7    int leak = *p;
8    printf("leak: %d\n",leak); }
9  }
10
```

Console

```
1  Unsuccessful termination of this exe
2    Undefined: [UB_CERB002a_out_of_bou
```

Memory

Zoom In | Zoom Out | Fit | Rese



Fundamental legacy problem: conventional architecture interface provides only coarse-grain mechanisms to enforce safety and security properties. Disastrous synergy with conventional C/C++ systems programming languages.

Many/most security vulnerabilities arise from memory-unsafety, but it is infeasible to radically change the industry-wide architectures or languages.

CHERI

CHERI

CHERI: research architecture to fundamentally improve security, with a hardware *capabilities* for fine-grained memory protection and sandboxing.

Originally (2010–): hardware/software co-design: PIs Robert Watson, Simon Moore (UCam); Peter Neumann (SRI).

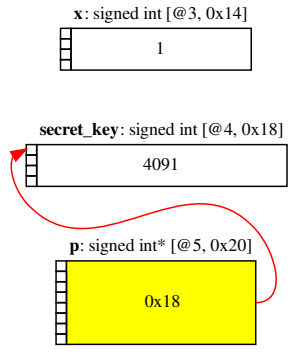
`https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/`

[Watson, Moore, Neumann, Sewell; Almatary, Anderson, Baldwin, Barrel, Bauereiss, Bukin, Chisnall, Clarke, Dave, Davis, Esswood, Filardo, Gudka, Gutstein, Joannou, Kovacsics, Laurie, Markettos, Maste, van der Maas, Mazzinghi, Mujumdar, Mundkur, Murdoch, Napierala, Nienhuis, Norton-Wright, Paeps, Paul-Trifu, Richardson, Roe, Rothwell, Rugg, Saidi, Son, Stolfa, Turner, Vadera, Woodruff, Xia, Zeeb]

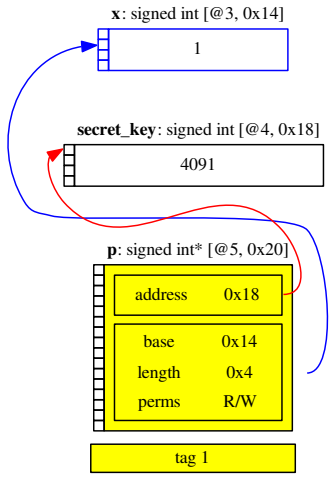
CHERI basic idea: add hardware support for capabilities

```
#include <stdio.h>
int x=1;
int secret_key = 4091;
int main() {
  int *p = &x;
  p = p+1;
  int y = *p;
  printf("%d\n",y);
}
```

ISO C



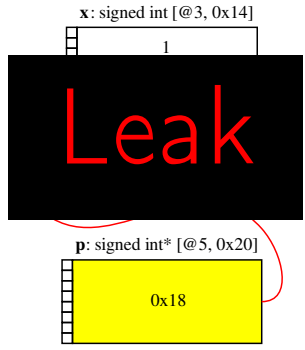
CHERI C



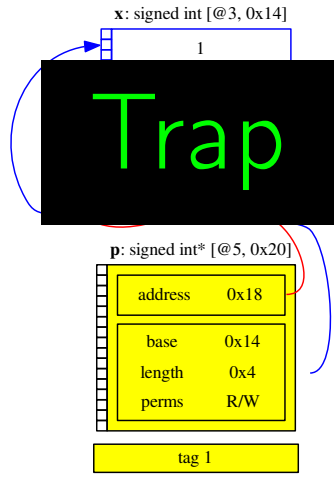
CHERI basic idea: add hardware support for capabilities

```
#include <stdio.h>
int x=1;
int secret_key = 4091;
int main() {
  int *p = &x;
  p = p+1;
  int y = *p;
  printf("%d\n",y);
}
```

ISO C



CHERI C



CHERI architecture key design points

- encoding allocation data and permissions within capability permits fast checking at access-time, without a lookup or TLB pressure
- ISA design lets code shrink capabilities, but never grow them
- non-addressable tags prevent forging (one bit per capability-sized/aligned unit of memory, cleared by any non-capability write, and one bit per register)
- compressed 128-bit encoding reduces extra memory cost
- can use capabilities either for all pointers, or just when desired
- co-exists nicely with existing C and C++
- co-exists nicely with existing virtual memory machinery (when desired)
- additional mechanisms (*sealed capabilities*) for secure encapsulation
- initial focus on *spatial* memory safety, but CHERI can also aid various *temporal* memory safety approaches

Scalable encapsulation using sealed capabilities

...omitted for today

CHERI architecture

Definitions of CHERI instruction-set architectures and their behaviour

First: CHERI-MIPS

Now: also exploring CHERI-RISC-V and (with Arm) CHERI-ARM

CHERI hardware

Hardware BlueSpec/FPGA CHERI-MIPS implementation, demonstrating that capabilities can be implemented efficiently:

- hierarchical tag cache for tagged memory
- efficient capability compression scheme
- avoid interference with conventional pipeline, MMU, etc.

Now: also CHERI-RISC-V

CHERI software

Have to demonstrate software performance and adaption costs are reasonable, and explore use of new protection mechanisms

Prototyped a complete software stack for CHERI by adapting widely used open-source software: Clang/LLVM, FreeBSD, FreeRTOS, and applications such as WebKit, OpenSSH, and PostgreSQL.

(also speaks to our ISO C work and v.v.)

Hardware/software co-design

Originally, with conventional engineering (mostly):

- prose+pseudocode ISA specification
- hand-written ISA test suite
- separate QEMU emulator

Successful, but painful!

And not much assurance of correctness – but that's essential for security

Semantics to the rescue?

previously: ISA semantics in HOL4 and L3 (Anthony Fox) (and for concurrency)

2014: shift to use formal L3 ISA semantics in CHERI-MIPS development

now: shifted to ISA semantics in Sail, for CHERI-MIPS and CHERI-RISC-V

- CHERI-MIPS: `https://github.com/CTSRD-CHERI/sail-cheri-mips`
- CHERI-RISCV: `https://github.com/CTSRD-CHERI/sail-cheri-riscv`

Rigorous Engineering

Lightweight:

- use formal ISA semantics as central design document
  (owned by CHERI researchers and engineers)
- use in architecture specification
  (readable)
- make executable as a test oracle, auto-translating L3/Sail to SML/OCaml/C
  ($\sim$ 400KIPS, booting FreeBSD in 4 min)
  - use for testing hardware against
  - use for software bring-up
    (supporting existing engineering practice)
- 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

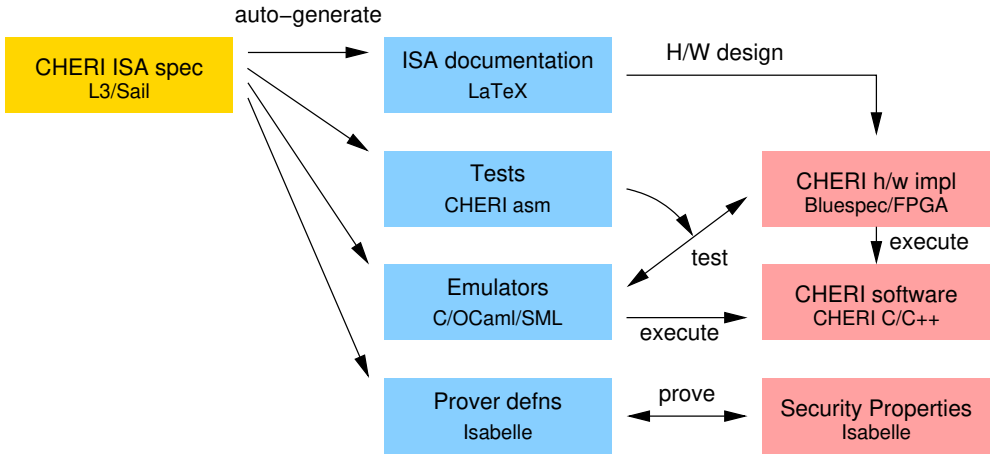
Rigorous Engineering

Heavyweight:

- auto-translate L3/Sail to HOL and Isabelle
- prove compression-scheme properties
- state (some of) the intended security properties of the CHERI-MIPS architecture
- prove them
- re-prove them

[Nienhuis, Joannou, Fox, Roe, Bauereiss, Campbell, Naylor, Norton, Moore, Neumann, Stark, Watson, Sewell; UCAM-CL-TR-940
`https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-940.pdf`]

The main artifacts of the CHERI engineering process



(plus CHERI-QEMU)

```
union clause ast = CLoad : (regno, regno, regno, bits(8), bool, WordType)
function clause execute (CLoad(rd, cb, rt, offset, signext, width)) = {
  checkCP2usable();
  let cb_val = readCapRegDDC(cb);
  if not (cb_val.tag) then
    raise_c2_exception(CapEx_TagViolation, cb)
  else if cb_val.sealed then
    raise_c2_exception(CapEx_SealViolation, cb)
  else if not (cb_val.permit_load) then
    raise_c2_exception(CapEx_PermitLoadViolation, cb)
  else {
    let 'size   = wordWidthBytes(width);
    let cursor  = getCapCursor(cb_val);
    let vAddr   = (cursor + unsigned(rGPR(rt)) + size*signed(offset)) % pow2(64);
    let vAddr64 = to_bits(64, vAddr);
    if (vAddr + size) > getCapTop(cb_val) then
      raise_c2_exception(CapEx_LengthViolation, cb)
    else if vAddr < getCapBase(cb_val) then
      raise_c2_exception(CapEx_LengthViolation, cb)
    else if not (isAddressAligned(vAddr64, width)) then
      SignalExceptionBadAddr(AdEL, vAddr64)
    else {
      let pAddr = TLBTranslate(vAddr64, LoadData);
      memResult : bits(64) = extendLoad(MEMr_wrapper(pAddr, size), signext);
      wGPR(rd) = memResult;
    } } }

function clause decode (0b110010 @ rd:regno @ cb:regno@ rt:regno @ offset:bits(8) @ 0b0 @ 0b00) = Some(CLoad(rd,cb,rt,offset,false,B)) /*CLBU*/
function clause decode (0b110010 @ rd:regno @ cb:regno@ rt:regno @ offset:bits(8) @ 0b1 @ 0b00) = Some(CLoad(rd,cb,rt,offset,true, B)) /*CLB*/
function clause decode (0b110010 @ rd:regno @ cb:regno@ rt:regno @ offset:bits(8) @ 0b0 @ 0b01) = Some(CLoad(rd,cb,rt,offset,false,H)) /*CLHU*/
function clause decode (0b110010 @ rd:regno @ cb:regno@ rt:regno @ offset:bits(8) @ 0b1 @ 0b01) = Some(CLoad(rd,cb,rt,offset,true, H)) /*CLH*/
function clause decode (0b110010 @ rd:regno @ cb:regno@ rt:regno @ offset:bits(8) @ 0b0 @ 0b10) = Some(CLoad(rd,cb,rt,offset,false,W)) /*CLWU*/
function clause decode (0b110010 @ rd:regno @ cb:regno@ rt:regno @ offset:bits(8) @ 0b1 @ 0b10) = Some(CLoad(rd,cb,rt,offset,true, W)) /*CLW*/
function clause decode (0b110010 @ rd:regno @ cb:regno@ rt:regno @ offset:bits(8) @ 0b0 @ 0b11) = Some(CLoad(rd,cb,rt,offset,false,D)) /*CLD*/
```

CHERI security properties

- define authority-inclusion order $\leq$ over capabilities
- define abstract capability intentions of each instruction
- characterise the capabilities a (potentially untrusted) compartment can access or construct by executing arbitrary code
- define *reachable capability monotonicity*
- define isolation assumptions and guarantees
- prove in Isabelle

Clarify some ambiguity along the way

[Nienhuis,...]

```
CompartmentIsolation sem is defined as
  for all addrs types s s′ trace step.
  if IsolatedState addrs types s
    and IntraDomainTrace trace
    and SwitchesDomain step
    and s′ ∈ FutureStates sem s (trace; step)
  then IsolationGuarantees addrs types s s′
```

```
IsolatedState addrs types s is defined as
  CapabilityAligned addrs
  and NoSystemRegisterAccess addrs types s
  and ContainedCapBounds addrs types s
  and ContainedObjectTypes addrs types s
  and InvokableCapsNotUsable addrs types s
  and not AccessToCU0 s
  and not KernelMode s
  and StateIsValid s
```

```
IsolationGuarantees addrs types s s′ is defined as
  Base (PCC s′) + PC s′
  ∈ ExceptionPCs ∪ InvokableAddresses addrs s
  and for all a.
      if not a ∈ TranslateAddresses addrs Store s
      then MemData s′ a = MemData s a
           and MemTag s′ (GetCapAddress a) =
               MemTag s (GetCapAddress a)
  and for all r.
      if r ≠ 0 and r ≠ 1 and r ≠ 31
      then SpecialCapReg s′ r = SpecialCapReg s r
```

T-CHERI

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

For capability-load, $[\![\texttt{CLW rd,cb,offset}]\!]$ is roughly the set of traces like:

$$[\texttt{E\_read\_reg}(\texttt{cb}, c),\ \texttt{E\_read\_mem}(\texttt{Read\_plain}, \mathit{addr}, 4, c'),\ \texttt{E\_write\_reg}(\texttt{rd}, c'')]$$

Axiom

If a tagged capability c is stored to memory at index i of a local trace t of an instruction, then c is derivable from capabilities that are available at index i of t.

[Bauereiss, Nienhuis, ...]

Hardware/software/semantics co-design!

Lightweight rigorous engineering methods all used by systems folk

Mechanised statements and proofs of security properties boost assurance in the architecture design

Practical evaluation, of performance and software adaption costs: looking good

Security evaluation: hard to do, but encouraging so far

Where next?

CHERI is a large academic project: 2010–date, ∼£24m DARPA+EPSRC+Industry (UCam)

Many papers, academically convincing evaluation.

Potential industry uptake?

Quite some enthusiasm, from major vendors – but CHERI touches the whole stack, even if only lightly. Really need industry-scale evaluation:

- port CHERI ideas to modern ISA (eg ARMv8-A, not MIPS)
- adapt high-performance out-of-order hardware implementation
- adapt more software
- invite others to experiment and evaluate

ISCF Digital Security by Design programme

Announced 2019-01: £70M government and £114M industry
Collaborators workshop: Thursday

"The InnovateUK ISCF Digital Security by Design challenge aims to radically update the foundation of the UK's insecure digital computing infrastructure."

- develop an industrial prototype/demonstrator for CHERI-ARM
  - define a prototype CHERI-ARM architecture (extending ARMv8-A)
    ...and make ISA semantics available
  - academic proofs of security properties
  - integrate into a high-end core
  - build a testchip SoC including that and tag caches etc
  - adapt software stacks (incl. Android)
  - integrate into a prototyping board (250–1000 units), made available for R&D by academics and major vendors
- support for academic and industry research around this (InnovateUK/EPSRC/ESRC)

ISCF Digital Security by Design programme

Goal is to generate compelling evidence to enable uptake, if possible

If successful, learnings from these will be adopted into future mainstream extensions to the Arm architecture (but no commitment to future compatability w.r.t. the prototype!).

More details from Arm on Thursday.

Enabling CHERI verification research

An Introduction to CHERI [Watson, Moore, Sewell, Neumann] (TR, this week)

Formal models available now:

- CHERI-MIPS
- CHERI-RISC-V
- ARMv8.5-A (with Arm)

All in Sail, and with generated Coq/HOL4/Isabelle prover models.

In future (with Arm):

- experimental CHERI-ARM
- security properties
- prover infrastructure for reasoning about the specification