Underpinning mainstream engineering with mathematical semantics

Peter Sewell
University of Cambridge

LMS/BCS-FACS Evening Seminar

18 November 2021

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.
Mainstream computer engineering

massively successful
elegant stack of abstractions
...but if we look straight

constant danger of collapse

huge problems from the legacy design choices

maintained only by constant security patching
Foundations?

Discrete Mathematics

Category Theory
Operational Semantics
Domain Theory
Denotational Semantics

Lambda Calculus
Type Systems
Type Theory
Complexity Theory
Computability Theory
Automata Theory

Process Calculi
Proof Assistants
SMT
Program Logics
Model Checking
Formal Specification

Dynamic Analysis
Static Analysis
Abstract Interpretation
Concurrency Theory

Logical Frameworks
Operational Semantics
Dynamic Analysis
Static Analysis
Abstract Interpretation
Discrete Mathematics

Lambda Calculus
Type Systems
Type Theory
Complexity Theory
Computability Theory
Automata Theory
Foundations?
Foundations?

test-and-debug development

prose specifications (at best)

HTTPS, TCP/IP
Linux, Windows
JavaScript, Python, Java
C, C++
Arm, x86
I don’t mean to imply that our existing theory has no connection to practice – much does.

But the fundamental abstractions that mainstream computing relies on, and the development processes most programmers use, remain mostly oblivious to the theory we have.

And, to a significant extent, vice versa.
Main Question

How can we develop mathematical semantics for the actual mainstream artifacts we rely on (not just idealised or toy versions, or semantics just about how we think the world ought to be) and apply it to improve the mainstream engineering of them?
Will look at this through the lens of several inter-related projects:

- network protocols
- relaxed-memory concurrency
- instruction-set architecture
- C
- CHERI
Part 1: NetSem – semantics of TCP and Sockets API
Motivation: existing protocol specifications (RFCs) are vague prose. Can we specify real protocols precisely with honest maths?

(Original: better failure semantics for process calculi)
NetSem: Semantics of network protocols – TCP/IP and the Sockets API

Engineering with Logic: Rigorous Test-Oracle Specification and Validation for TCP/IP and the Sockets API

STEVE BISHOP, University of Cambridge† († when this work was done), UK
MATTHEW FAIRBAIRN, University of Cambridge†, UK
HANNES MEHNERT, robur.io, Center for the Cultivation of Technology, Germany
MICHAEL NORRISH, Data61, CSIRO and Australian National University, Australia
TOM RIDGE, University of Leicester, UK
PETER SEWELL, University of Cambridge, UK
MICHAEL SMITH, University of Cambridge†, UK
KEITH WANSBROUGH, University of Cambridge†, UK

Conventional computer engineering relies on test-and-debug development processes, with the behaviour of common interfaces described (at best) with prose specification documents. But prose specifications cannot be used in test-and-debug development in any automated way, and prose is a poor medium for expressing complex (and loose) specifications.

The TCP/IP protocols and Sockets API are a good example of this: they play a vital role in modern communication and computation, and interoperability between implementations is essential. But what exactly they are is surprisingly obscure: their original development focussed on “rough consensus and running code”, augmented by prose RFC specifications that do not precisely define what it means for an implementation to be correct. Ultimately, the actual standard is the de facto one of the common implementations, including, for example, the 15000–20000 lines of the BSD implementation — optimised and multithreaded C code, time-dependent, with asynchronous event handlers, intertwined with the operating system, and security-critical.

This paper reports on work done in the Netsem project to develop lightweight mathematically rigorous techniques that can be applied to such systems: to specify their behaviour precisely (and formally enough to assist the required implementation evolution) and to test whether them

Success: defined in HOL4 an envelope of allowed behaviour for TCP/IP and Sockets, with clear and experimentally testable relationship to production impls.

...can cope with real artifacts, without idealisation or much restriction

...learned a lot about experimental semantics, testing, and working at scale
NetSem: Approach

*Experimental* approach: validating semantics against existing implementations (the de facto standards)

Forms of semantics:
- paper – supports hand proof
- mechanised
  - in a prover – supports mechanised proof
  - executable – can run as a (maybe slow) implementation
- *executable as a test oracle* – can decide whether some observable behaviour is allowed or not
NetSem: Approach

Experimental approach: validating semantics against existing implementations (the de facto standards)

Forms of semantics:
- paper – supports hand proof
- mechanised
  - in a prover – supports mechanised proof
  - executable – can run as a (maybe slow) implementation
  - *executable as a test oracle* – can decide whether some observable behaviour is allowed or not

Need the last:
- to validate semantics against implementations (without full impl correctness proof)
- to test implementations against semantics – to make directly usable for engineers in standard test-and-debug development
NetSem: Approach

Experimental approach: validating semantics against existing implementations (the de facto standards)

Forms of semantics:
- paper – supports hand proof
- mechanised
  - in a prover – supports mechanised proof
  - executable – can run as a (maybe slow) implementation
  - executable as a test oracle – can decide whether some observable behaviour is allowed or not

Need the last:
- to validate semantics against implementations (without full impl correctness proof)
- to test implementations against semantics – to make directly usable for engineers in standard test-and-debug development

Key technical challenge: achieving this in the face of specification looseness/nondeterminism
For TCP/IP: did so with custom symbolic evaluation in HOL4
NetSem: Reflections

Working at scale: 9k LoS, 9 person-years. More like code than classic maths.

Have to deal both with
- “fundamental” systems complexity – it’s a subtle protocol, for good engineering reasons
- contingent complexity, from historical accidents and mistakes

To be useful, need both! Not just some idealisation enough for a single paper.
NetSem: Reflections

Failure? Stopped too soon:

...didn’t produce turnkey system, usable by practitioners

...used fancy tools, making that hard (in hindsight, could be much simpler)

...didn’t influence standards

...didn’t get to point of proving substantial theorems about the protocol

...problem wasn’t sufficiently widely appreciated?
Part 2: Relaxed-memory concurrency
Relaxed-memory concurrency

2007: Susmit Sarkar arrives as new postdoc in Cambridge

We want to work on hypervisor verification (Xen)

...but what’s the underlying hardware programming model?
2007: Susmit Sarkar arrives as new postdoc in Cambridge

We want to work on hypervisor verification (Xen)

...but what’s the underlying hardware programming model?

...impossible to tell
What is relaxed-memory concurrency?

Naively, shared-memory concurrency has *sequentially consistent* semantics:

Threads are interleaved in some arbitrary sequential order, consistent with program order within each thread, with each read reading from the most recent write to the same location.

Most theory has assumed that.
What is relaxed-memory concurrency?

In reality, mainstream architectures and programming languages give much weaker guarantees for low-level concurrent code. Consider e.g.

<table>
<thead>
<tr>
<th>MP+ctrl</th>
<th>Pseudocode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td>Thread 1</td>
</tr>
<tr>
<td>x=1</td>
<td>while (r1=y)==0 {</td>
</tr>
<tr>
<td>y=1</td>
<td>r2=x</td>
</tr>
<tr>
<td>Initial state: x=0, y=0</td>
<td>Final: 1:r1=1, 1:r2=0</td>
</tr>
</tbody>
</table>

Observable (and allowed) on ARMv7, Armv8-A, IBM Power, and RISC-V (not on x86)

So you can’t reason in terms of a simple global-time model

Test MP+ctrl
What is relaxed-memory concurrency?

In reality, mainstream architectures and programming languages give much weaker guarantees for low-level concurrent code. Consider e.g.

<table>
<thead>
<tr>
<th>MP+ctrl</th>
<th>Pseudocode</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread 0</strong></td>
<td><strong>Thread 1</strong></td>
</tr>
<tr>
<td>x=1</td>
<td>while ((r1=y)==0) {}</td>
</tr>
<tr>
<td>y=1</td>
<td>r2=x</td>
</tr>
<tr>
<td>Initial state: x=0, y=0</td>
<td>Final:1:r1=1, 1:r2=0</td>
</tr>
</tbody>
</table>

Test MP+ctrl

Observable (and allowed) on ARMv7, Armv8-A, IBM Power, and RISC-V (not on x86)

So you can’t reason in terms of a simple global-time model

Why? Observable effects of microarchitecture and compiler optimisations:
- pipeline: out-of-order and speculative execution
- storage subsystem: write `propagation` in either order
- compiler: common subexpression elimination
How do we tell what behaviour is allowed?

Approach #1: examine vendor architecture documentation of the time
How do we tell what behaviour is allowed?

Approach #1: examine vendor architecture documentation of the time

“Principle 5. Intel 64 memory ordering ensures transitive visibility of stores — i.e. stores that are causally related appear to execute in an order consistent with the causal relation”  
[Intel White Paper, 2007]
How do we tell what behaviour is allowed?

Approach #1: examine vendor architecture documentation of the time

IBM Power barriers (ARMv7 text similar):

“For each applicable pair $a_i, b_j$ of storage accesses such that $a_i$ is in A and $b_j$ is in B, the memory barrier ensures that $a_i$ will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before $b_j$ is performed with respect to that processor or mechanism.

- A includes all applicable storage accesses by any such processor or mechanism that have been performed with respect to P1 before the memory barrier is created.
- B includes all applicable storage accesses by any such processor or mechanism that are performed after a Load instruction executed by that processor or mechanism has returned the value stored by a store that is in B.”
How do we tell what behaviour is allowed?

Approach #1: examine vendor architecture documentation of the time

IBM Power barriers (ARMv7 text similar):

“For each applicable pair $a_i$ and $b_j$ of storage accesses such that $a_i$ is in A and $b_j$ is in B, the memory barrier ensures that $a_i$ will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before $b_j$ is performed with respect to that processor or mechanism.

- A includes all applicable storage accesses by any such processor or mechanism that have been performed with respect to P1 before the memory barrier is created.
- B includes all applicable storage accesses by any such processor or mechanism that are performed after a Load instruction executed by that processor or mechanism has returned the value stored by a store that is in B.”

Part 2: Relaxed-memory concurrency
How do we tell what behaviour is allowed?

Approach #1: examine vendor architecture documentation of the time

- presumes the documentation expresses a coherent model
Approach #2: look at previous research, 1978–2008


...and language-level work, especially around JMM and early work towards C/C++11 (Manson, Pugh, Boehm, Adve)
Approach #2: look at previous work, 1978–2008

x86 (Intel, AMD)  
IBM Power  
ARMv7  
ARMv8  
RISC-V  
SPARC TSO  
SPARC RMO, PSO  
Itanium  
MIPS  
Alpha  

?  
?  
?  
didn’t exist yet  
didn’t exist yet  
✓  
✓  
✓  
✓  
?  

gone

Part 2: Relaxed-memory concurrency
Approach #3
Approach #3

- experiment (following TCP ideas)
Approach #3

- experiment (following TCP ideas)
  - build hardware test harness, litmus
    (Maranget; early versions by Braibant and Zappa Nardelli, and by Sarkar)
  - make models *executable as test oracles* in various tools
  - hand-write and auto-generate libraries of interesting tests
    (auto-generation with diy, Alglave & Maranget)
  - find various hardware bugs in production silicon along the way
Approach #3

- experiment (following TCP ideas)
  - build hardware test harness, litmus
    (Maranget; early versions by Braibant and Zappa Nardelli, and by Sarkar)
  - make models executable as test oracles in various tools
  - hand-write and auto-generate libraries of interesting tests
    (auto-generation with diy, Alglave & Maranget)
  - find various hardware bugs in production silicon along the way

- talk with vendors (mainly IBM + Arm) to find out what they intend
  - hard to find right people and get their time
  - presumes they know what they mean (for architectural envelope, not just what they build)
    - prose specs didn’t give them the tools to think about that clearly
  - but their architectural intent is primary
Approach #3

- experiment (following TCP ideas)
  - build hardware test harness, litmus
    (Maranget; early versions by Braibant and Zappa Nardelli, and by Sarkar)
  - make models *executable as test oracles* in various tools
  - hand-write and auto-generate libraries of interesting tests
    (auto-generation with diy, Alglave & Maranget)
  - find various hardware bugs in production silicon along the way

- talk with vendors (mainly IBM + Arm) to find out what they intend
  - hard to find right people and get their time
  - presumes they know what they mean (for architectural envelope, not just what they build)
    - prose specs didn’t give them the tools to think about that clearly
  - but their architectural intent is primary

- iterate: compare models and hardware, relate to language models, prove things, discuss with vendors, work with RISC-V TG, refine models
Approach #3

- experiment (following TCP ideas)
  - build hardware test harness, litmus
    (Maranget; early versions by Braibant and Zappa Nardelli, and by Sarkar)
  - make models *executable as test oracles* in various tools
  - hand-write and auto-generate libraries of interesting tests
    (auto-generation with diy, Alglave & Maranget)
  - find various hardware bugs in production silicon along the way

- talk with vendors (mainly IBM + Arm) to find out what they intend
  - hard to find right people and get their time
  - presumes they know what they mean (for architectural envelope, not just what they build)
    – prose specs didn’t give them the tools to think about that clearly
  - but their architectural intent is primary

- iterate: compare models and hardware, relate to language models, prove things, discuss with vendors, work with RISC-V TG, refine models

hence (working with IBM, Arm, RISC-V) *create* precise architectural abstractions

Architectural relaxed-memory concurrency, 2021

- x86 (Intel, AMD) ✔ user: de facto
- IBM Power ✔ user: de facto
- ARMv8-new ✔ user: official system: in progress
- RISC-V ✔ user: official
- SPARC TSO ✔

- ARMv8-old ✔ ish
- ARMv7 ✔ ish
- SPARC RMO, PSO ✔ not used
- Itanium ✔ going
- Alpha ✔ gone
- MIPS ?
What kind of semantics do we end up with?

1. *Abstract-microarchitectural operational models*

   abstract machines that capture the hardware intuition, to explain what’s going on, with explicit out-of-order and speculative execution, but without the complexity of real hardware. Incrementally executable
Thread semantics: out-of-order, speculative execution abstractly

Our thread semantics has to account for out-of-order and speculative execution.

- Instructions can be fetched before predecessors finished
- Instructions independently make progress
- Branch speculation allows fetching successors of branches
- Multiple potential successors can be explored

NB actual hardware implementations can and do speculate even more, e.g. beyond strong barriers, so long as it is not observable
Operational model

- each thread has a tree of instruction instances;
- no register state;
- threads execute in parallel above a flat memory state: mapping from addresses to write requests
- for Power: need more complicated memory state to handle non-MCA

(For now: plain memory reads, writes, strong barriers. All memory accesses same size.)
Commit Barrier

**Condition:**
A barrier instruction \( i \) in state Plain (\( \text{Barrier}({\text{barrier\_kind}}, {\text{next\_state}}) \)) can be committed if:

1. all po-previous conditional branch instructions are finished;
2. (BO) if \( i \) is a \text{dmb} \text{ sy} instruction, all po-previous memory access instructions and barriers are finished.

Really: functional code that computes allowed transitions
What kind of semantics do we end up with?
What kind of semantics do we end up with?

Part 2: Relaxed-memory concurrency
What kind of semantics do we end up with?
What kind of semantics do we end up with?

Part 2: Relaxed-memory concurrency
What kind of semantics do we end up with?
What kind of semantics do we end up with?

2. *Axiomatic*: predicates on candidate complete execution graphs

typically forbidding certain cycles using various relations over those graphs. More concise, further from hardware intuition, not incrementally executable
Example: speculative execution

![Diagram of speculative execution]

**Allowed.** The edges form a cycle, but `ctrl; [R]` to read events is not in `ob`.

```plaintext
acyclic pos | fr | co | rf

let obs = rfe | fre | coe

let dob = addr | data |
        | ctrl; [W]
        | addr; po; [W]
        | (ctrl | data); coi
        | (addr | data); rfi
...

let bob = po; [dmb.sy]; po
...

let ob = obs | dob | aob | bob

acyclic ob
```

Content: 4.5 Armv8-A, IBM Power, and RISC-V: Armv8-A/RISC-V axiomatic model
What kind of semantics do we end up with?
Write forwarding from an unknown-address write

**PPOAA**

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
</table>
| a:
W x=1   | d:
R y=1   |
| fen  | rfe  |
| c:
W y=1 | e:
W z=1 |
| fre | po |
| rf | rfi |
| g:
R x=0 |   |

Let

- `obs = rfe | fre | coe`
- `dob = addr | data | ctrl; [W] | addr; po; [W] | (ctrl | data); coi | (addr | data); rfi`
- `bob = po; [dmb.sy]; po`

`acyclic ob`

Forbidden. ob includes addr; rfi: forwarding is only possible when the address is determined.

Contents 4.5 Armv8-A, IBM Power, and RISC-V: Armv8-A/RISC-V axiomatic model 393

Part 2: Relaxed-memory concurrency 51
Write forwarding on a speculative path

**PPOCA**

Thread 0

- W x=1 a:
- W y=1 c:
- Thread 1
- R y=1 d:
- W z=1 e:
- R z=1 f:
- R x=0 g:

**Allowed.** Forwarding is allowed: rfi (and ctrl;rfi and rfi;addr) not in ob (compare x86-TSO)

**Acyclic**

- pos | fr | co | rf
- let obs = rfe | fre | coe
- let dob = addr | data
  | ctrl; [W]
  | addr; po; [W]
  | (ctrl | data); coi
  | (addr | data); rfi
- ...
- let bob = po; [dmb.sy]; po
- ...
- let ob = obs | dob | aob | bob

Contents 4.5 Armv8-A, IBM Power, and RISC-V: Armv8-A/RISC-V axiomatic model
What kind of semantics do we end up with?

3. *Promising-Arm Operational* (more abstract, not much like hardware – but still incrementally executable) [Pulte, Pichon-Pharabod, Kang, Lee, Hur]

All useful, for different purposes

Proved equivalent:

- Abstract-microarchitectural and axiomatic [Pulte]
How do we make the semantics executable as test oracles?

**Operational:**
- rmem tool [Flur, Pulte, French, Gray, Sarkar, Sewell, ...]
  exhaustively find all abstract-machine executions
  (optimising to avoid exploring trivial interleavings)

**Axiomatic:**
- herd tool [Alglave, Maranget]
  exhaustively find all candidate execution graphs (optimised), and check axiomatic-model
  predicate for each
- isla-axiomatic [Armstrong, Simner, Campbell]
  use SMT solver on combination of axiomatic model and Isla symbolic execution of ISA
Arm systems concurrency: ifetch, IC, DC, etc. in isla-axiomatic

Part 2: Relaxed-memory concurrency
Arm systems concurrency: relaxed virtual memory

TLB caching needs explicit synchronisation: on Armv8-A, break-before-make. This is the ‘break’ side of break-before-make, but without an ISB at the end on the same thread, so it is not guaranteed that the po-later translations for this core are restarted.

Part 2: Relaxed-memory concurrency
Major contributors for our joint x86, IBM Power, ARM h/w models, chronologically. See also much other work by Alglave et al., by Lustig and other RISC-V TG members, and by others for GPUs, transactions, persistence.
Programming-language relaxed-memory concurrency

Similar but harder problem: union of hardware models, plus compiler optimisations

Worked with ISO C++ WG21 Concurrency Group to formalise and partially fix the design for C/C++11 concurrency

[Mark Batty, Scott Owens, Susmit Sarkar, Tjark Weber, Peter Sewell; and later by others]

Axiomatic model, based on DRF-SC

Executable-as-test oracle semantics in cppmem and Cerberus-BMC

Experiment much harder here – but proof of implementation schemes (not full impls!) more feasible
Relaxed-memory reflections

Success:

- helped clarify what behaviour these major industry abstractions allow (x86, Armv8-A, IBM Power, RISC-V, C, C++; JavaScript, WebAssembly)
- revealing implicit complexity contributed to substantial revisions of some of these
- mathematical relaxed-memory semantics and tools based on it now routinely used in industry
- theory and verification researchers can build on these models (and fix them where needed), with program logics, model-checking, models for other features, ...
Relaxed-memory reflections

- addressing an area where industry *knew* it had problems
Relaxed-memory reflections

- addressing an area where industry *knew* it had problems
- established credibility through finding early specification ambiguities (asking hard to-the-point questions) and finding hardware bugs
Relaxed-memory reflections

- addressing an area where industry knew it had problems
- established credibility through finding early specification ambiguities (asking hard to-the-point questions) and finding hardware bugs
- had to take seriously industry incentives and concerns, balancing with academic publication
Relaxed-memory reflections

- addressing an area where industry knew it had problems
- established credibility through finding early specification ambiguities (asking hard to-the-point questions) and finding hardware bugs
- had to take seriously industry incentives and concerns, balancing with academic publication
- invented semantics as appropriate – turned out quite unlike traditional concurrency semantics
Relaxed-memory reflections

- addressing an area where industry knew it had problems
- established credibility through finding early specification ambiguities (asking hard to-the-point questions) and finding hardware bugs
- had to take seriously industry incentives and concerns, balancing with academic publication
- invented semantics as appropriate – turned out quite unlike traditional concurrency semantics
- emphasis on understanding what the architectural abstraction is, not on traditional formal verification of implementations, or on bug-finding (though we did find some h/w bugs, and prove correctness of implementation schemes)
Relaxed-memory reflections

- addressing an area where industry knew it had problems
- established credibility through finding early specification ambiguities (asking hard to-the-point questions) and finding hardware bugs
- had to take seriously industry incentives and concerns, balancing with academic publication
- invented semantics as appropriate – turned out quite unlike traditional concurrency semantics
- emphasis on understanding what the architectural abstraction is, not on traditional formal verification of implementations, or on bug-finding (though we did find some h/w bugs, and prove correctness of implementation schemes)
- as for TCP, had to take seriously both fundamental and contingent complexity of existing artifacts
Relaxed-memory reflections

- addressing an area where industry knew it had problems
- established credibility through finding early specification ambiguities (asking hard to-the-point questions) and finding hardware bugs
- had to take seriously industry incentives and concerns, balancing with academic publication
- invented semantics as appropriate – turned out quite unlike traditional concurrency semantics
- emphasis on understanding what the architectural abstraction is, not on traditional formal verification of implementations, or on bug-finding (though we did find some h/w bugs, and prove correctness of implementation schemes)
- as for TCP, had to take seriously both fundamental and contingent complexity of existing artifacts
- blending science and engineering
Relaxed-memory reflections

- addressing an area where industry knew it had problems
- established credibility through finding early specification ambiguities (asking hard to-the-point questions) and finding hardware bugs
- had to take seriously industry incentives and concerns, balancing with academic publication
- invented semantics as appropriate – turned out quite unlike traditional concurrency semantics
- emphasis on understanding what the architectural abstraction is, not on traditional formal verification of implementations, or on bug-finding (though we did find some h/w bugs, and prove correctness of implementation schemes)
- as for TCP, had to take seriously both fundamental and contingent complexity of existing artifacts
- blending science and engineering
- had to make semantics and tools that are mature and useful, not just enough for a paper – in it for the long haul
Relaxed-memory reflections

- addressing an area where industry knew it had problems
- established credibility through finding early specification ambiguities (asking hard to-the-point questions) and finding hardware bugs
- had to take seriously industry incentives and concerns, balancing with academic publication
- invented semantics as appropriate – turned out quite unlike traditional concurrency semantics
- emphasis on understanding what the architectural abstraction is, not on traditional formal verification of implementations, or on bug-finding (though we did find some h/w bugs, and prove correctness of implementation schemes)
- as for TCP, had to take seriously both fundamental and contingent complexity of existing artifacts
- blending science and engineering
- had to make semantics and tools that are mature and useful, not just enough for a paper – in it for the long haul
- invest research and engineering effort in proportion to the problem size
Part 3: Instruction-set architecture semantics
Instruction-set architecture (ISA) semantics

Architecture semantics

≈ Concurrency model subtle but small, as above
  + Instruction-set-architecture (ISA) semantics more straightforward but large
  + other system-on-chip (SoC) semantics ...ignore still
Instruction-set architecture (ISA) semantics

ISA semantics traditionally paper pseudocode (at best), in vendor manuals

Armv8-A
- Alastair Reid et al. made the Arm-internal ASL mechanised, within Arm
- we translate it into our custom ISA definition language, Sail
- and thence to provers and C, validating against AVS
- 100k LoS

RISC-V
- we hand-wrote an ISA semantics in Sail
- adopted by RISC-V International as their official formal spec
- 10k LoS

Both complete enough to boot an OS or hypervisor, and pretty authoritative
Sequential Emulator (C)
Sequential Emulator (OCaml)
Test Generation
Concurrent Execution
Concurrency models
Axiomatic, Cat
Concurrency models
Operational, Lem
Concurrency models
Axiomatic, Cat
Concurrency models
Operational, Lem
ISA Security Properties
(Machine-checked proofs)

Part 3: Instruction-set architecture semantics
Different kind of problem: not asking what the semantics is
...but rather, how can we make it practical to work with these large formal definitions
...not just for researchers, but also for conventional engineers

Solution: Sail language design and engineering
not much nondeterminism in sequential spec
...so can make executable as test oracle “just” with translations to C and OCaml
Part 4: C Semantics
C Semantics

C semantics

≈ Sequential thread-local semantics subtle and large, but largely in ISO
  + Memory object model subtle and small; unknown
  + Relaxed memory model subtle and small; see above

Cerberus C semantics: Kayvan Memarian, Victor Gomes, Stella Lau, Kyndylan Nienhuis, Justus Matthiesen, Peter Sewell

More complex interaction with industry, via ISO C/C++ committees and Clang/GCC communities
Part 5: CHERI
Computers are still terrible (even if we can define them more precisely)

Those legacy design choices:

- systems languages that don’t enforce protection (C/C++)
- hardware that only enforces coarse-grain protection, using virtual memory

Baked in to the critical systems codebase and the entire industry.

Result, in today’s adversarial environment:

- programming errors can often lead to exploitable vulnerabilities
- 50%+ (?) of security problems involve a memory safety violation
Cerberus C semantics:

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

int x = 1;
int secret_key = 4091;

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

int x = 1;
int secret_key = 4091;

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

int x = 1;
int secret_key = 4091;

int main() {
    int *p = &x;
    p = p + 1;
    int leak = *p;
    printf("leak: %d\n", leak);
}
```
So what happens if we compile and run it?
$ gcc -std=c11 -O0 -Wall -pedantic 35c3.c
$ ./a.out
leak: 4091
How can we (practically) make things better?

How can we use semantics to increase assurance and ease engineering, at design time rather than post facto?
CHERI Security

CHERI: architecture extensions to support *hardware-enforced*
- fine-grain memory protection
- secure encapsulation
using unforgeable capabilities

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

Hardware/software/semantics co-design since 2014

- Main CHERI page: www.cheri-cpu.org
#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);
}
CHERI basic idea: add hardware support for capabilities

ISO C

```c
#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);
}
```

CHERI C

```c
#include <stdio.h>

int x: signed int [@3, 0x14]

int secret_key: signed int [@4, 0x18]

int main() {
    int *p = &x;
    p = p+1;
    int y = *p;
    printf("%d\n",y);
}
```

UB/Exploit

Part 5: CHERI
CHERI basic idea: add hardware support for capabilities

ISO C

```c
#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);
}
```

CHERI C

```
#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);
}
```
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 ("pure cap"), or just when desired
- co-exists nicely with existing C and C++, existing ISAs, existing virtual memory (when desired)
- additional sealed capabilities for secure encapsulation (skip today)
- initial focus was on spatial memory safety, but CHERI also supports various temporal memory safety approaches (skip today)
CHERI C/C++ fine-grained pure-capability protection

This is using capabilities instead of integer pointers throughout an address space, protecting from exploitable coding errors.

Initial power-on universal capability successively refined by the kernel, run-time linker, compiler-generated code, heap allocator, ...

Spatial protection automatically applied at two levels:

**Language-level pointers** to stack and heap allocations, global variables, TLS variables, subobjects

**Language implementation pointers** stack pointers, return addresses, C++ vtable pointers, GOT pointers, PLT entry pointers, vararg array pointers, ELF aux arg pointers, ...

‘nice’ C/C++ gets all this just from a recompile
How about interaction between mutually untrusting components?

Classic virtual memory protection ok for processes, but doesn’t scale well, e.g. for browser tabs, mail-readers, and server-side code handling untrusted data, with controlled sharing

Use CHERI capabilities (including the sealing mechanism and/or default capabilities) for this, e.g. to encapsulate instances of untrusted libraries, and for co-processes – a different way of using CHERI (and can combine with fine-grain protection)

No need to update MMU mappings, so very low domain-crossing cost
Does it work?

Software porting cost: ranging down to 0.04% LoC for well-behaved C

Vulnerabilities: MSRC estimate large fraction of their critical vulnerabilities would be deterministically mitigated

Performance: encouraging – but hard to know for sure from academic studies
CHERI People

Robert N. M. Watson¹, Simon W. Moore¹, Peter Sewell¹, Peter G. Neumann⁷

Arm: Graeme Barnes², Richard Grisenthwaite², Lee Eisen², and many more

Hesham Almatary¹, Jonathan Anderson¹*, Alasdair Armstrong¹, Peter Blandford-Baker¹, John Baldwin⁷, Hadrien Barrel¹, Thomas Bauereiss¹, Ruslan Bukin¹, Brian Campbell³, David Chisnall¹*,¹¹, Jessica Clarke¹, Nirav Dave⁷*, Brooks Davis⁷, Lawrence Esswood¹, Nathaniel W. Filardo¹*,¹¹, Franz Fuchs¹, Khilan Gudka¹*, Brett Gutstein¹, Alexandre Joannou¹, Robert Kovacsics¹*, Ben Laurie⁵, A. Theo Markettos¹, J. Edward Maste¹*, Alfredo Mazzinghi¹, Alan Mujumdar¹*, Prashanth Mundkur⁷, Steven J. Murdoch¹*, Edward Napierala¹, Kyndylan Nienhuis¹, Robert Norton-Wright¹*,¹¹, Philip Paeps¹*, Lucian Paul-Trifu¹*, Allison Randal¹, Ivan Ribeiro¹, Alex Richardson¹*,⁵, Michael Roe¹, Colin Rothwell¹*, Peter Rugg¹, Hassen Saidi⁷, Thomas Sewell¹, Stacey Son¹*, Ian Stark³, Domagoj Stolfa¹*, Andrew Turner¹, Munraj Vadera¹*, Jonathan Woodruff¹, Hongyan Xia¹*, Vadim Zaliva¹, Bjoern A. Zeeb¹*,

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, * previously

Part 5: CHERI
Academic CHERI

SRI + Cambridge over 11 years + 3 DARPA programs (≈$26M), EPSRC (£7.4M); Innovate UK (£2.7M); Google / DeepMind / Arm / HPE ... (≈£1M)

Architecture design:
  ▶ CHERI-MIPS, CHERI-RISC-V

Hardware implementations (BSV/FPGA):
  ▶ CHERI-MIPS and extensions of BSV RISC-V cores (Piccolo, Flute, Toooba)
  ▶ ...and Qemu emulator

Software stack:
  ▶ LLVM, linker, FreeBSD, FreeRTOS, temporal safety, ...

Semantics:
  ▶ lightweight and heavyweight “rigorous engineering”
Lightweight:

- use formal ISA semantics, in L3 and Sail, as central design documents (owned by CHERI researchers and engineers)
- use in architecture specification (readable)
- make executable as a test oracle, auto-translating Sail/L3 to C/OCaml/SML (~ 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
- use for automatic test generation
- auto-translate to SMT and use to check properties
What does CHERI guarantee?

Q. But how do we know the architecture *does* enforce the intended security protections?

Q. How can we even *state* them precisely?
What does CHERI guarantee?

Q. But how do we know the architecture does enforce the intended security protections?

Q. How can we even state them precisely?

A. Use L3 and Sail infrastructure to enable machine-checked mathematical proof that specific precise properties always hold.
CHERI security properties

Theorem

For any intra-domain trace, the reachable capabilities from the final state are no greater than those of the initial state.

Theorem

Any trace within a properly set-up compartment cannot affect other memory, and can exit the compartment only in controlled ways.

Properties of arbitrary code above the CHERI-MIPS ISA.

Mechanised Isabelle proofs above L3 model


Adapted proofs as ISA evolved ("regression proof")

("Heavyweight Rigorous Engineering" – but not hw or sw verif)
Central question for adoption: does CHERI provide good protection at acceptable performance and software-porting cost?

Academic evaluation very encouraging – Arm and others interested and involved – but it’s not evaluated in a modern high-end superscalar core

Hard for industry to commit without that – but hard to get industry-scale evidence without major investment in demonstrator
5-year Digital Security by Design UKRI program: £70M UK gov. funding, £117M industrial match, to create CHERI-Arm (“Morello”) prototype architecture, hardware implementation, demonstrator SoC + board, software, and proofs

Leap over that supply-chain gap that makes adopting new architecture difficult – validating concepts in microarchitecture, architecture, and software at scale

Arm, UCam, U.Ed., Linaro, and additional industrial and academic R&D (EPSRC, ESRC, Innovate UK)

2020 emulation models; 2021/2022 Morello board delivery.
5-year Digital Security by Design UKRI program: £70M UK gov. funding, £117M industrial match, to create CHERI-Arm ("Morello") prototype architecture, hardware implementation, demonstrator SoC + board, software, and proofs

Leap over that supply-chain gap that makes adopting new architecture difficult – validating concepts in microarchitecture, architecture, and software at scale

Arm, UCam, U.Ed., Linaro, and additional industrial and academic R&D (EPSRC, ESRC, Innovate UK)

2020 emulation models; 2021/2022 Morello board delivery. As of last month: pure-capability CheriBSD kernel boots on silicon!
The Morello Board

- An Industrial Demonstrator of a Capability architecture
- Uses a prototype capability extension to the Arm Architecture
  - Prototype is a “superset” of what could be adopted into the Arm architecture
- Use of a superset of the architecture is very unusual
  - Also unrealistic as a commercial product – there will be some frequency effects
  - However, there are tight timescales so architecture is nearly complete now
- The superset of the architecture will allow a lot of software experimentation
  - Various different mechanisms for compartmentalisation
  - Collection of features for which the justification is unclear
  - Techniques for holding the capability tag bit
- Architecture will have formally proved security properties (with UoC and UoE)
- Morello Board will be the ONLY physical implementation of this prototype architecture
  - Learnings from these experiments will be adopted into a mainstream extension to the Arm architecture
  - NO COMMITMENT TO FULL BINARY COMPATIBILITY TO THE PROTOTYPE ARCHITECTURE
  - But successful concepts are expected to be carried forward into the architecture and can be reused there
Morello Board overview (subject to change)

- Quad core bespoke high-end CPU with prototype capability extensions
  - Backwards compatibility with v8.2 AArch64-only
  - Based on Neoverse N1 core
    - Multi-issue out-of-order superscalar core with 3 levels of cache
  - Build in 7nm process
  - Targeting clock frequency around 2GHz

- Reasonable performance GPU and Display controller
  - Standard Mali architecture core – not extended with capability
  - Supports Android

- PCIe and CCIx interfaces including to FPGA based accelerators
- FPGA for peripheral expansion
- SBSA compliant system
- 16GB of System Memory (expandable to 32GB – tbc)
Morello software stacks

Complete open-source CHERI-enabled software stack from bare metal up, to validate and evaluate design, and support future R&D:

<table>
<thead>
<tr>
<th>Open-source application suite (WebKit, Python, OpenSSH, nginx, PostgresQL, ...)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CheriBSD/Morello (SRI/Cambridge)</strong></td>
</tr>
<tr>
<td>▶ FreeBSD kernel + userspace, application stack</td>
</tr>
<tr>
<td>▶ Kernel spatial and referential memory protection</td>
</tr>
<tr>
<td>▶ Userspace spatial, referential, and temporal memory protection</td>
</tr>
<tr>
<td>▶ Intra-process compartmentalization</td>
</tr>
<tr>
<td>▶ Co-process IPC</td>
</tr>
<tr>
<td>▶ Armv8-A 64-bit binary compatibility for legacy binaries</td>
</tr>
<tr>
<td><strong>Android (Arm)</strong></td>
</tr>
<tr>
<td><strong>CHERI-extended Google Hafnium hypervisor (Morello only)</strong></td>
</tr>
<tr>
<td><strong>CHERI Clang/LLVM compiler suite, LLD, LLDB, GDB</strong></td>
</tr>
</tbody>
</table>
Morello ISA verification

Morello ISA designed by Arm, with detailed discussion. In ASL, extending Armv8-A

We [Bauereiss, Campbell, Thomas Sewell, Armstrong]:

- 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

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

Conclusion
(How) can we develop mathematical semantics for the actual mainstream artifacts we rely on, and apply it to improve the mainstream engineering of them?

Yes, we can

How?
Conclusion: back to the main question

(How) can we develop mathematical semantics for the actual mainstream artifacts we rely on, and apply it to improve the mainstream engineering of them?

Yes, we can

How?

- (partly) exploit advances in computational power and proof tools
(How) can we develop mathematical semantics for the actual mainstream artifacts we rely on, and apply it to improve the mainstream engineering of them?

Yes, we can

How?

- (partly) exploit advances in computational power and proof tools
- (mostly) focus on interfaces
(How) can we develop mathematical semantics for the actual mainstream artifacts we rely on, and apply it to improve the mainstream engineering of them?

Yes, we can

How?

- (partly) exploit advances in computational power and proof tools
- (mostly) focus on interfaces
- (especially where there’s a need for more clarity)
(How) can we develop mathematical semantics for the actual mainstream artifacts we rely on, and apply it to improve the mainstream engineering of them?

Yes, we can

How?

- (partly) exploit advances in computational power and proof tools
- (mostly) focus on interfaces
- (especially where there’s a need for more clarity)
- ...on executable-as-test-oracle semantics for them,
Conclusion: back to the main question

(How) can we develop mathematical semantics for the actual mainstream artifacts we rely on, and apply it to improve the mainstream engineering of them?

Yes, we can

How?

- (partly) exploit advances in computational power and proof tools
- (mostly) focus on interfaces
- (especially where there’s a need for more clarity)
- …on executable-as-test-oracle semantics for them,
- …inventing new semantics as appropriate, and
(How) can we develop mathematical semantics for the actual mainstream artifacts we rely on, and apply it to improve the mainstream engineering of them?

Yes, we can

How?

- (partly) exploit advances in computational power and proof tools
- (mostly) focus on interfaces
- (especially where there’s a need for more clarity)
- ...on executable-as-test-oracle semantics for them,
- ...inventing new semantics as appropriate, and
- ...on engaging with mainstream artifacts and engineering processes
(How) can we develop mathematical semantics for the actual mainstream artifacts we rely on, and apply it to improve the mainstream engineering of them?

Yes, we can

How?

▷ (partly) exploit advances in computational power and proof tools
▷ (mostly) focus on interfaces
▷ (especially where there’s a need for more clarity)
▷ ...on executable-as-test-oracle semantics for them,
▷ ...inventing new semantics as appropriate, and
▷ ...on engaging with mainstream artifacts and engineering processes

And with collaboration with many excellent colleagues