End-to-End Formal Verification of a RISC-V Processor Extended with Capability Pointers

Dapeng Gao Tom Melham

University of Oxford

The problem

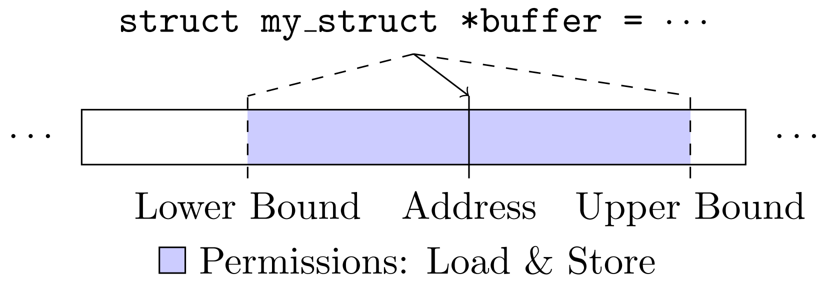
- Systemic security flaws in conventional computer architectures
  - Unsafe memory
    ‘70% of the vulnerabilities addressed through a security update each year continue to be memory safety issues’ [Miller, BlueHat IL 2019]
- Solution: CHERI, a security extension to conventional ISAs
  - Automatically mitigates 31% of reported vulnerabilities [MSRC 2020]
    - Only a conservative estimate—can go up to 67% when using advanced features
  - Verification of CHERI hardware is essential to guarantee security

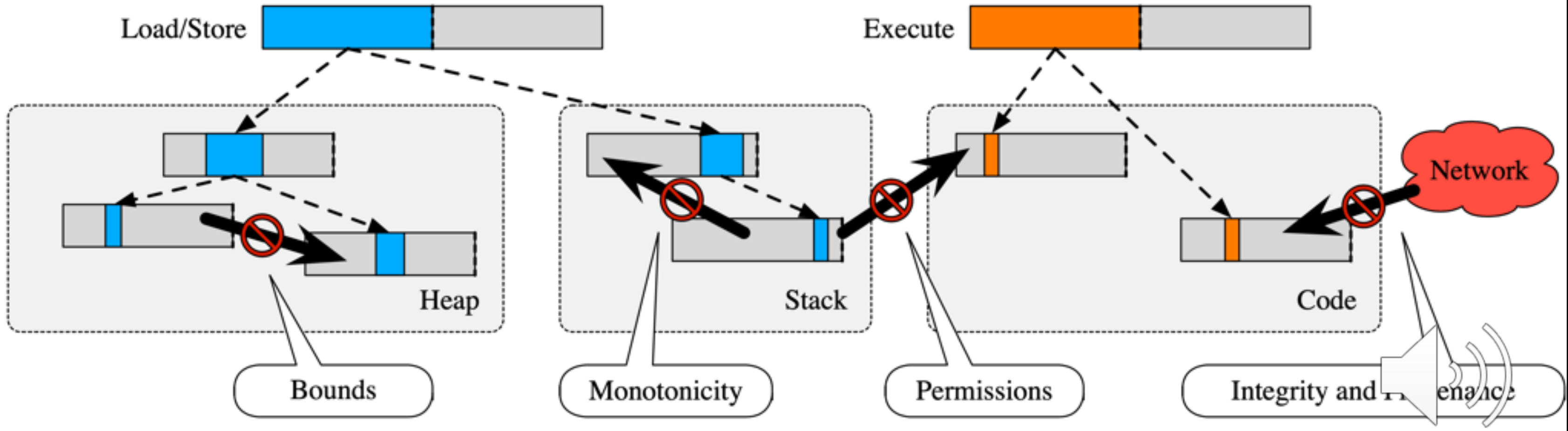
This talk

- First extensive formal verification of a CHERI processor
- Novel proof engineering methodologies

Capability Hardware Enhanced RISC Instructions (CHERI)

- Replace integer-based pointers by 128-bit wide ‘capabilities’
  - Compressed lower and upper bounds limit accessible memory
  - Read/Write/Execute permissions and more limit authorised operations
  - 1-bit validity tag (stored out-of-band) for each capability location
  - Can only derive less privileged capabilities from more privileged ones
  - Throws exception deterministically if any of the above is violated





Source: CHERI ISAv8

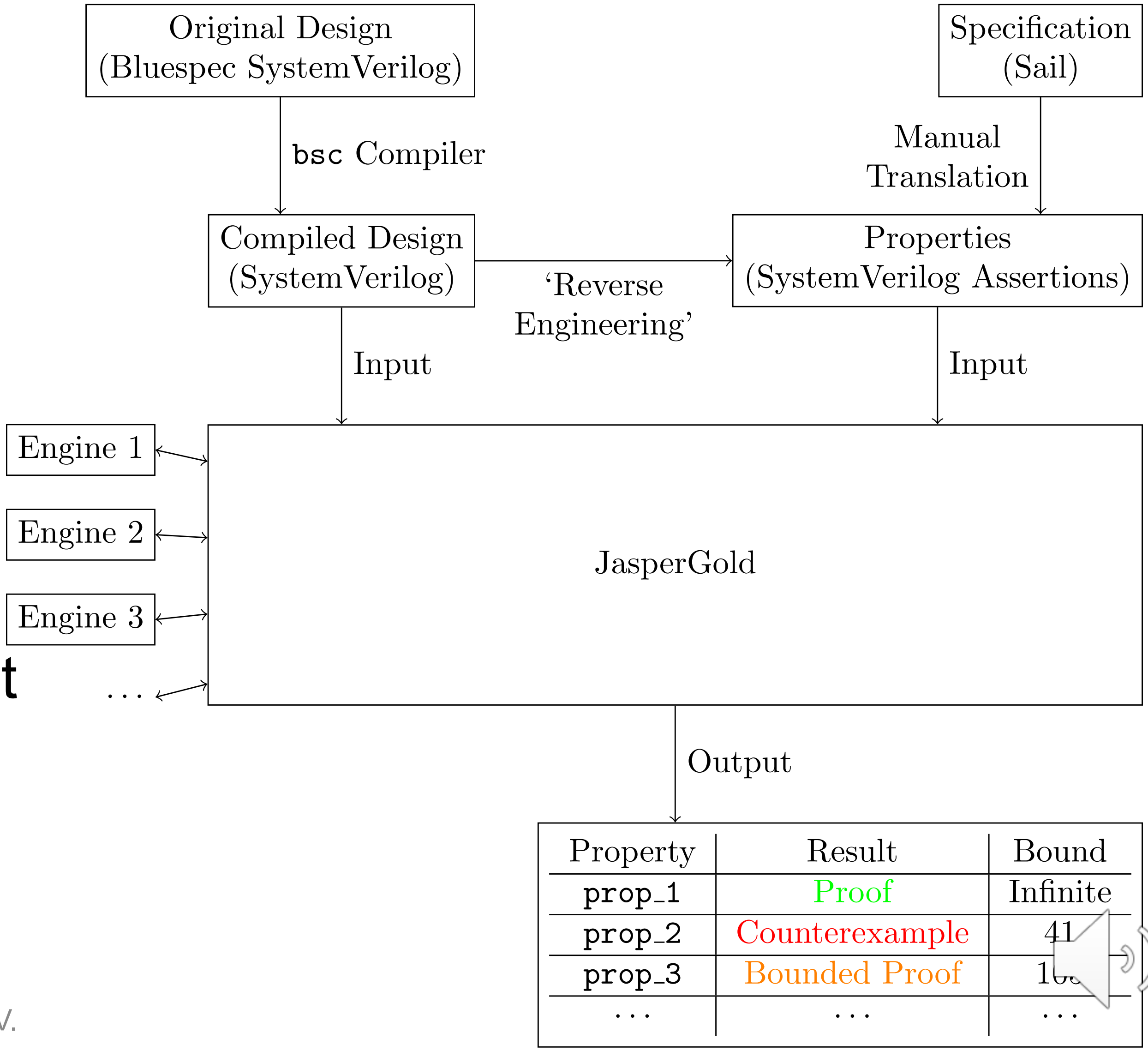
CHERI protection

- Mitigates many classes of memory-related exploits
  - Buffer overflow
  - Control flow redirection
- Advanced features
  - In-process software compartmentalisation
- ISA extension for secure capability manipulation, e.g.

• CIncOffset	Increments address, clears tag if it goes out-of-bound
• CSetBounds	Reduces the bounds
• CAndPerm	Reduces the permissions

Verification flow

- CHERI-Flute[†] is an open-source five-stage processor
  - Written in Bluespec SystemVerilog (BSV)
  - Compiles to SystemVerilog (SV)
- JasperGold only supports SV
  - Need to ‘reverse-engineer’ the compiled SV to identify relevant signal names
- Specification written in Sail
  - Manual translation into SV



Property	Result	Bound
prop_1	Proof	Infinite
prop_2	Counterexample	41
prop_3	Bounded Proof	10[illegible]
. . .	. . .	. . .

[†] Modified from the original Flute processor that implements plain RISC-V.

Simplified Example: CSetBounds

- Initial state A_n = {*regfile*: …, *pc*: …, etc.}
- Execute CSetBounds *cd*, *cs1*, *rs2* †
  - Reduce the length of the bounds of *cs1* to *rs2* and then write the result to *cd*—successful retirement
  - However, if either
    - *cs1* is *invalid* (i.e. validity tag is unset), or
    - *rs2* is *greater* than the length of the bounds of *cs1*,

    } Guard conditions
  - Then an exception is thrown—unsuccessful retirement
- Final state A_{n+1}
- How to formulate this as a property of the *microarchitecture*?

† Notation: Register names beginning with 'c' (e.g. *cs1*) denote capability registers. Those beginning with 'r' (e.g. *rs1*) denote integer registers.

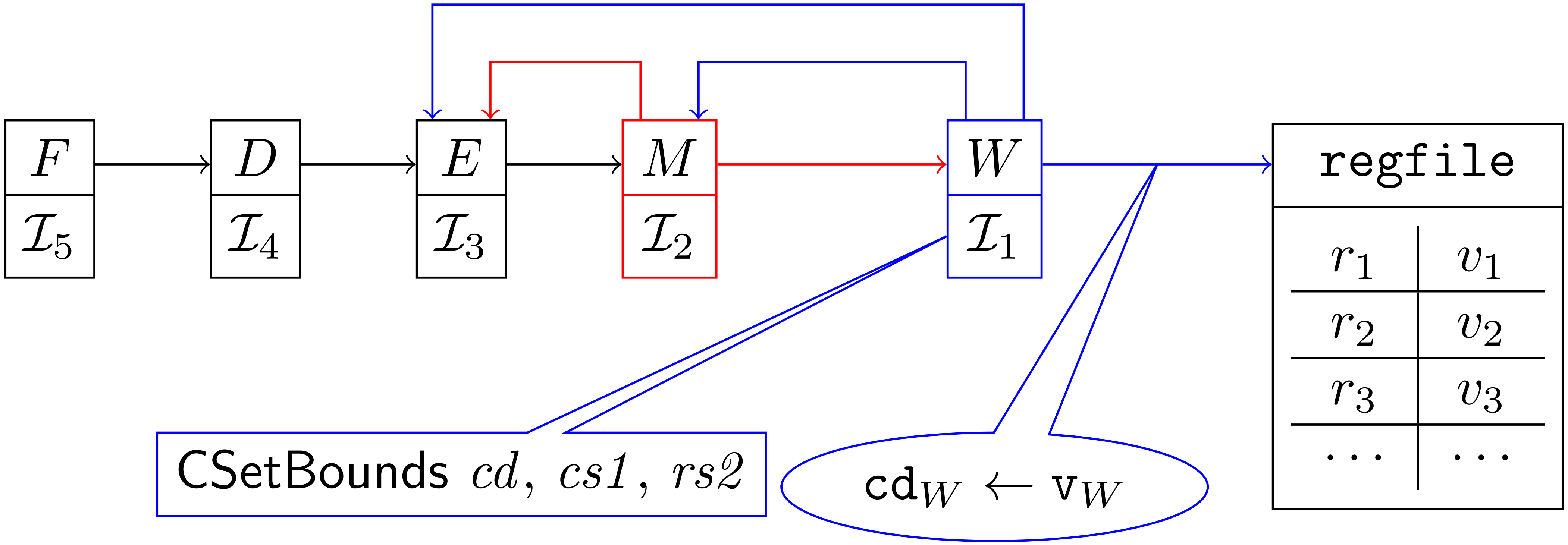
Formulating specification as properties (1/3)

- Let α be an abstraction function that maps each microarchitectural state to an architectural state
- To verify instruction *I*, assert: For any *microarchitectural* state *s*,
  - If retiring instruction *I* brings the processor from state *s* to *s'*, then
  - According to the specification, executing instruction *I* should alter the *architectural* state from α(*s*) to α(*s'*)
- This is difficult to implement in practice
  - An instruction can retire successfully in only *one* way
  - But it can retire unsuccessfully in *many* ways
  - Difficult to capture all possible outcomes in manually-written properties

Formulating specification as properties (2/3)

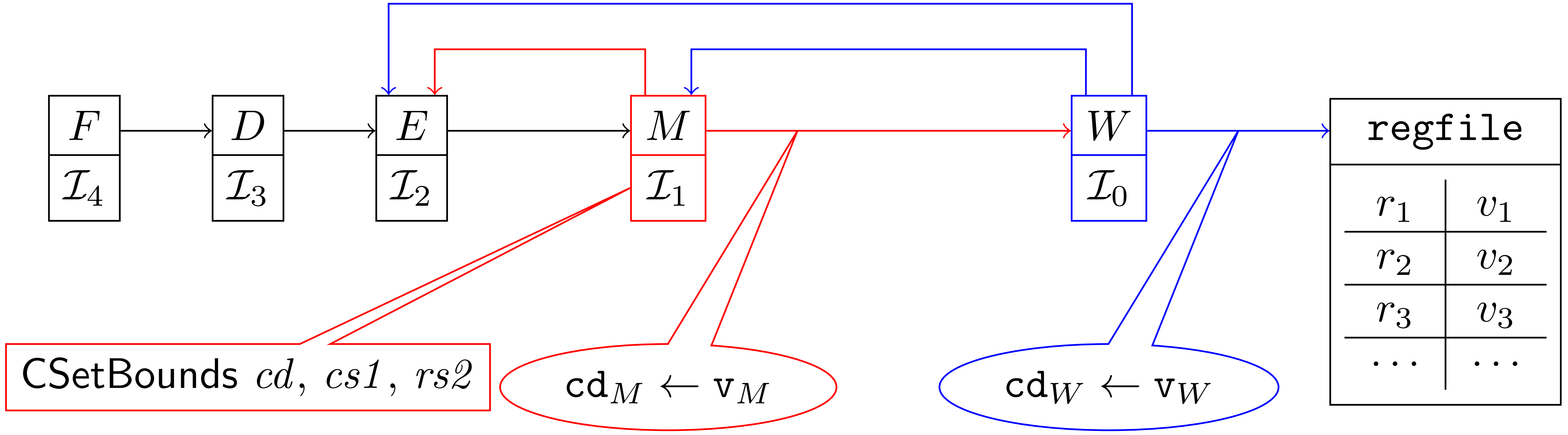
- Weakened to: For any microarchitectural state *s*,
  - If successfully retiring instruction *I* brings the processor from state *s* to *s'*, then
  - According to the specification, executing instruction *I* alters the architectural state from α(*s*) to α(*s'*) and all instruction *I*'s guard conditions are met
- Contrapositive implies: If any of instruction *I*'s guard conditions are *not* met, then instruction *I* is *not* successfully retired
  - Still enforces the security guarantees offered by CHERI

Formulating specification as properties (3/3)



- When I_1 is in stage W (i.e. about to be retired),
  - $\texttt{v}_W = \mathit{result}_{\text{CSetBounds}}(\texttt{regfile}[\mathit{cs1}], \texttt{regfile}[\mathit{rs2}])$, and
  - $\mathit{guard}_{\text{CSetBounds}}(\texttt{regfile}[\mathit{cs1}], \texttt{regfile}[\mathit{rs2}])$
- Unfortunately, proof engines do not converge for this property

Proof engineering (1/2): Decomposing the pipeline



- Define $\texttt{regfile}_M[r] = (\text{if } \texttt{cd}_W = r \text{ then } \texttt{v}_W \text{ else } \texttt{regfile}[r])$
- When I_1 is in stage M (and about to move to stage W),
  - $\texttt{v}_M = \mathit{result}_{\text{CSetBounds}}(\texttt{regfile}_M[\mathit{cs1}], \texttt{regfile}_M[\mathit{rs2}])$, and
  - $\mathit{guard}^{M}_{\text{CSetBounds}}(\texttt{regfile}_M[\mathit{cs1}], \texttt{regfile}_M[\mathit{rs2}])$

Proof engineering (2/2): Developing microarchitectural invariants

- CHERI instructions execute sophisticated functions in the ALU
  - Efficient bounds-checking and address computation
- Sail specification can handle potentially malformed inputs
- Hardware only works for well-formed capabilities
  - Malformed capabilities can never be created in the first place
- Model checker unaware that malformed capabilities never occur
  - Creates unreachable counterexamples in SAT-based model checking
- Use *k*-induction to prove global consistency invariant
  - State-Space Tunnelling (SST) is used to achieve convergence

Also covered in paper

- Correctness properties for
  - Branching instructions
  - Memory instructions
- Liveness properties
  - With the help of fairness constraints

Results (1/4)

- End-to-end verification of 80+ CHERI instructions
  - Final properties are basically agnostic of the microarchitecture
- Black-boxed
  - Cache and memory
  - Branch predictor
- Template-based properties
  - One template for each class of instructions
  - Can instantiate properties for different instructions with a few templates
  - Allows quick testing of new proof ideas on large number of instructions

Results (2/4)

- Discovered several edge case bugs
  - Some ALU functions can yield malformed capabilities
  - Some CSR registers are not cleared during reset
  - The CUnseal instruction does not clear a permission bit
  - The AUIPCC instruction incorrectly clears the validity tag
  - The CSetAddr instruction returns incorrect results
    - Rare corner case—escaped previous simulation-based testing
    - Easily uncovered by formal verification
- All bugs have been confirmed by the designers

Results (3/4)

- Bug or Feature? More design-side investigation is required
- An instruction gives better results than the specification
  - Perhaps the specification should be changed instead
- Corrupted memory can introduce malformed capabilities
  - Sanitise all capabilities read from memory?

Results (4/4)

- Pattern of bug discovery
  1. Try to prove property
  2. If proof fails, report the counterexample
  3. If proof does not converge, use SST to generate a ‘counterexample’
  4. Usually, the ‘counterexample’ involves a malformed capability and should be unreachable, so we try to strengthen the global consistency invariant

Conclusion and prospects

- Need automatic translation from the Sail specification to SVA
  - Can eliminate the need to weaken the specification
- Can serve as basis for verifying complex out-of-order designs
  - Morello—a high-performance CHERI-enabled prototype board by Arm
  - Eventually apply in verifying commercial CHERI processors