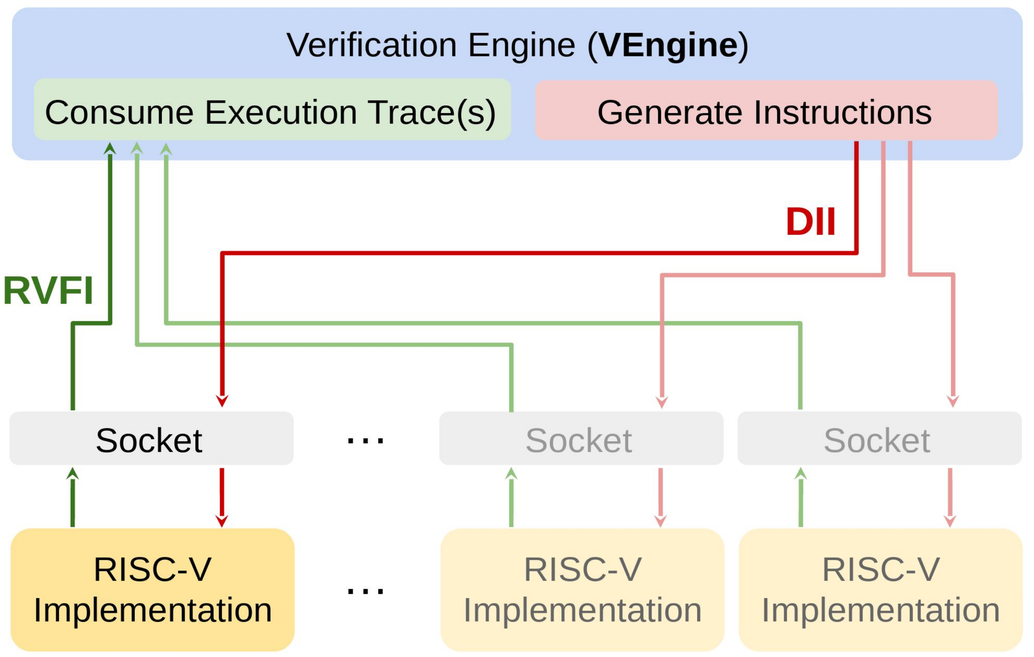
Who tests the TestRIG? Tooling for randomised tandem verification

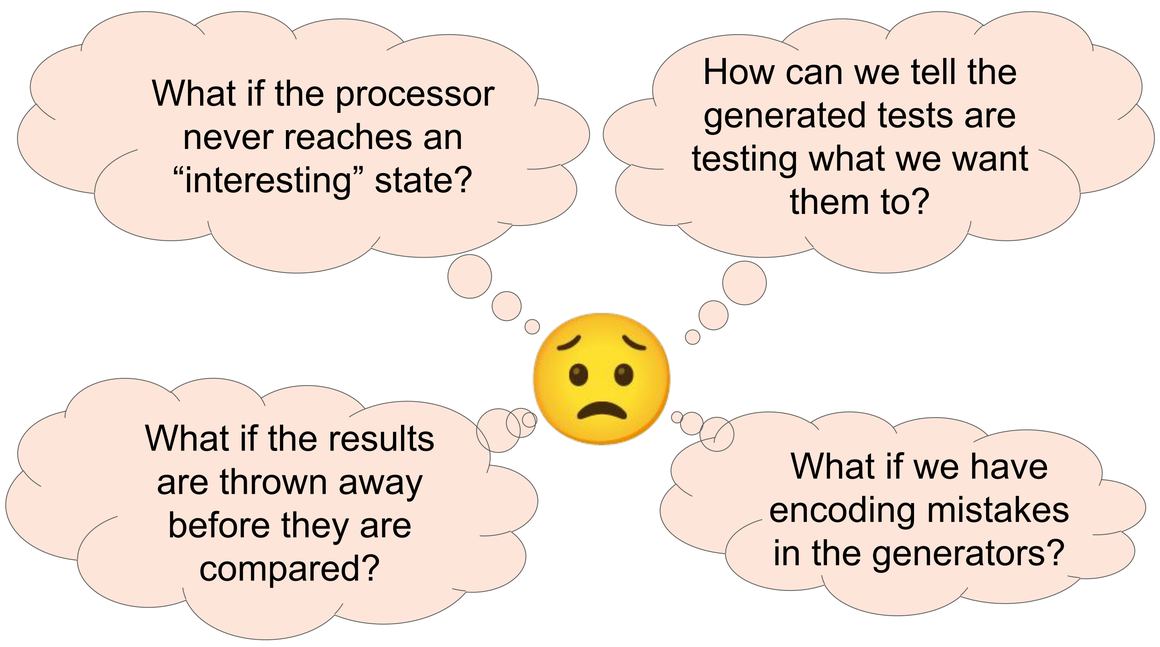
Peter Rugg, Alexandre Joannou, Jonathan Woodruff, Franz A. Fuchs, and Simon W. Moore
Department of Computer Science and Technology, University of Cambridge
peter.rugg@cl.cam.ac.uk
Project URL: cheri-cpu.org

UNIVERSITY OF CAMBRIDGE
Computer Science & Technology

TestRIG

TestRIG is an ecosystem for cross-verifying RISC-V implementations using a standard **RVFI-DII interface**. Verification Engines connect to the implementations over this interface. **QuickCheckVEngine** uses Haskell's QuickCheck library to **generate tests** and **automatically shrink** any divergences to a minimal reproducer. The RISC-V **golden Sail model** implements RVFI-DII, allowing implementations to be compared against this (hopefully) correct-by-definition executable simulator.
Since initial publication, the TestRIG infrastructure has seen increasing community engagement, including users and contributors from **Microsoft Research**, **lowRISC**, and **SCI Semiconductor**. The repository now links to 10 RVFI-DII-extended implementations, and has several forks from other members of the community.
TestRIG is in use to test **CHERI**, which adds unforgeable hardware capabilities for memory safety and compartmentalisation, in the **Toooba** and **CVA6** processors. The RISC-V community is in the process of **standardising** CHERI extensions.





Measuring Coverage

Attempt: code coverage

Measuring lines of code run is a good start, but has the problem that code may be run, but have **no effect** on the tested outputs, causing coverage to pass but **bugs to be missed**. For example, CHERI checks may get run on capabilities where the integrity tag is already clear, **hiding the error.**

Solution: mutation coverage

- **Simulate real bugs** by modifying the Sail code with incorrect behaviour
- **Definitively** answer: "would the test framework have caught that bug?"
- Automatically try classes of **common mistakes**

Implementation

- **Python classes** describe **mutation types**: pattern and transformation
- Coverpoints and run results are tracked in a **SQLite Database**
- Support for building and running many mutants in **parallel**
- Produces pretty **HTML reports** to highlight coverage issues

```
function clause execute (RISCV_JAL(imm, rd)) = {
  let target = PC + sign_extend(imm);
  /* Perform standard alignment check */
  let target_bits = bits_of(target);
  if bit_to_bool(target_bits[1]) then
  {
    RETIRE_FAIL
  } else {
    X(rd) = get_next_pc();
    set_next_pc(target_bits);
    RETIRE_SUCCESS
  }
}
```

Sail JAL semantics
(simplified for illustration)

Automatic mutation

```
function clause execute (RISCV_JAL(imm, rd)) = {
  let target = PC + sign_extend(imm);
  /* Perform standard alignment check */
  let target_bits = bits_of(target);
  if false then
  {
    RETIRE_FAIL
  } else {
    X(rd) = get_next_pc();
    set_next_pc(target_bits);
    RETIRE_SUCCESS
  }
}
```

Mutant buggy semantics

TestRIG

```
blt x20, x0, -1954
bgeu x1, x19, 496
auipc x1, 504870
blt x19, x2, -3472
jalr x3, x0, 842
bge x3, x0, 1220
bge x18, x0, 1280
bne x16, x16, 2934
auipc x0, 614292
jal x0, -225406
jal x20, 757650
blt x1, x17, 2414
jalr x20, x18, -951
beq x3, x2, -3610
jal x17, 128270
beq x19, x16, 3294
jal x16, -823264
beq x20, x3, -2520
beq x1, x18, -1072
beq x16, x0, -3224
```

✅ covered

Long counterexample trace

Automatic shrinking

```
Trap:  True, PCWD: 0x0000000000000000, RD: 00, RWD: 0x0000000000000000, I: 0x50e1f8ef PRV_M XL:32 (jal x17, 128270)
^ A, B v: mismatch in field trap: 1 != 0, mismatch in field pc_wdata: 0x0 != 0x8001f50e
Trap: False, PCWD: 0x000000008001f50e, RD: 17, RWD: 0x0000000080000004, I: 0x50e1f8ef PRV_M XL:32 (jal x17, 128270)
```

Minimal, single instruction annotated reproducer

Currently supported mutation types:

- Delete "encdec" mappings ➡️ "Every instruction tested"
- Delete code lines ➡️ "Every side effect tested"
- Replace branch conditions ➡️ "Every behaviour tested"

Useful **artifacts** to collect for a **compatibility test suite**

Future Work

- Switch from Python text transformations to directly interacting with the **Sail compiler**
- Automatically **inline** Sail functions
- **Extend** supported mutations
- Compare effectiveness of **different test suites**
- Identify and close coverage **blind-spots**
- Extend to **microarchitectural** mutation coverage in a Verilog implementation
- Find lots of **bugs**?

Bonus: relaxing the "tandem"

Aligning implementation and model on every possible instruction input is hard! CSR legalisation, store conditional failures, misaligned accesses,...
TestRIG now supports a "**single implementation**" mode: run implementation on its own and assert more **liberal properties** over RVFI trace.
Example property: instruction count in = instruction count out, i.e. processor did not **lockup**.
Template to inject **undirected** random instructions found and diagnosed:

- Processor **mis-decoded** and locked up on certain illegal instructions
- Subtle and rare compressed branch mispredict **infinite loop**
- Reachable **fatal assert** condition in a version of the Sail model

DII is required to allow such liberal testing without having to reason about control flow.

SRI International

CHERI