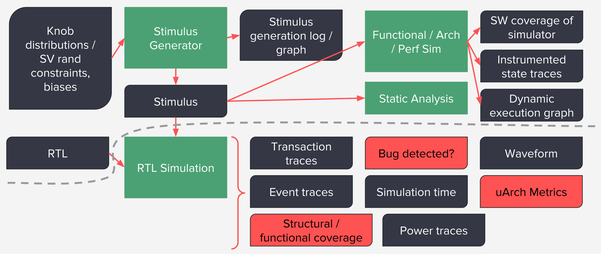
Parametric Stimulus Generators for Controllable RTL Fuzzing

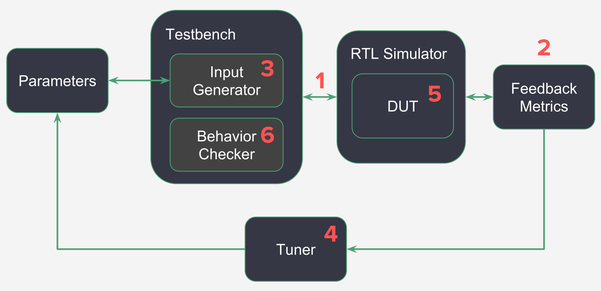
Rohit Agarwal, Vighnesh Iyer, Bora Nikolic
UC Berkeley

Motivation



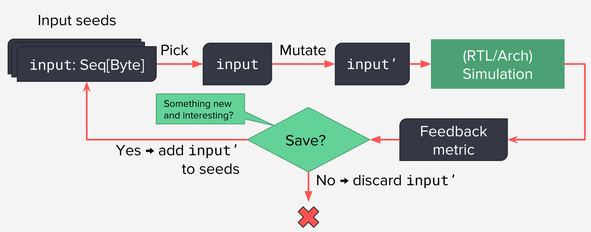
- RTL simulation is the workhorse of and golden reference for uArch evaluation
- Good stimulus generation is critical to hit desired post-RTL-simulation metrics, but is time-consuming and manual
- Fuzzing can theoretically generate stimulus to target any user-defined metric or for bughunting
- *How do we build good stimulus generators for hardware fuzzing?*

The Dynamic RTL Simulation Environment



1. **Testbench API**: interface between testbench and RTL simulator (SystemVerilog, SimCommand, cocotb)
2. **Feedback metrics**: can include coverage and/or uArch metrics to guide simulation
3. **Input generator**: produces stimulus for DUT
4. **Tuner**: adjusts generator parameters based on feedback (usually a human)
5. **Bug inducers**: circuit modifications that make bugs more likely to surface (e.g. backpressure randomization)
6. **Behavior checker**: DUT-specific, golden models or temporal properties

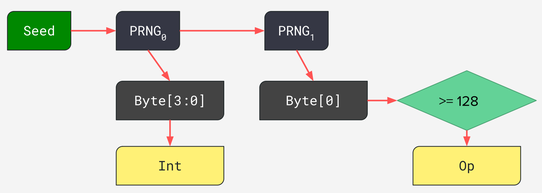
Adapting SW Fuzzing for HW



- DUT ports driven directly from sequences of bytes
- New inputs created via *random mutation*
- Naive fuzzing produces illegal stimulus and fails to reach interesting DUT states
- An *input generator* can produce legal and structural stimulus

PRNG-Based Input Generators

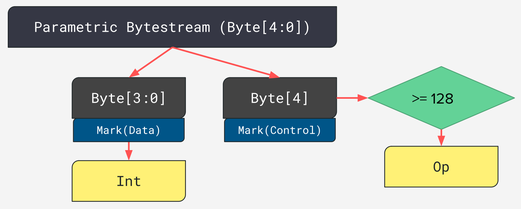
```
enum Op { read, write }
Gen[(Int, Op)].uniform().generate()
```



- Each random decision is pulled from a PRNG
- We can only control the seed! No precise or semantic control over the stimulus

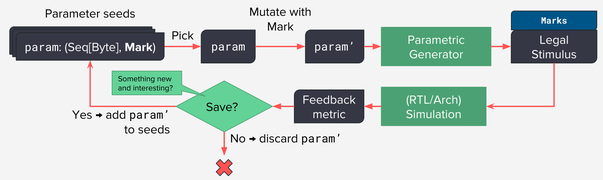
Parametric Input Generators

```
enum Op { read, write }
Gen[(Int, Op)].uniform().generate(Seq(1, 2, 3, 4, 199))
[3:0] -> Data, [4] -> Control
```



- "Random" decisions come from a user-provided bytestream
- We gain precise control over the stimulus
- "Marks" annotate bytes with how they are used in stimulus construction

Parametric Hardware Fuzzing



- Parametric generators give the mutator fine control over the semantics of the legal stimulus it generates
- Marks guide mutation rates of different bytes

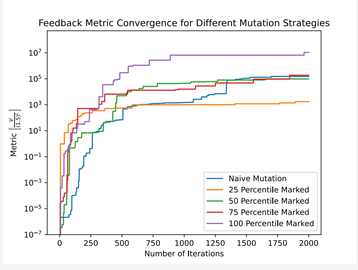
Generative Parametric Random Stimulus API

- A functional API for random stimulus generation that is backed by a source of randomness or a controllable bytestream

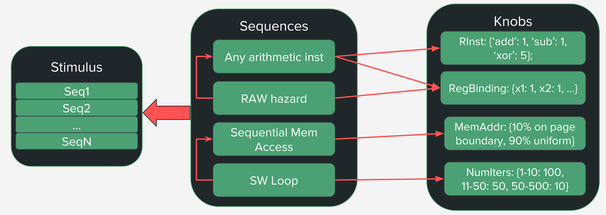
```
enum SExpr:
  case Expression(operator: Op, e1: SExpr, e2: SExpr)
  case Num(n: Int)

val maxDepth = 10
val maxNum = 30
/* Makes calculator expressions recursively. */
def genSExpr(depth: Int = 0): Gen[SExpr] =
  for {
    goDeeperProb ← Gen.double.mark(Struct)
    expr ← if (goDeeperProb < 1 - depth / maxDepth)
      then for {
        op ← genOperator.mark(Op)
        exp1 ← genSExpr(depth + 1)
        exp2 ← genSExpr(depth + 1)
      } yield (Expression(op, exp1, exp2))
      else for {
        n ← Gen.range(0, maxNum).mark(Num)
      } yield (Num(n))
  } yield (expr)
```

Mark-Driven Mutation For SExpr Generation

- Mutating different types of bytes with different probabilities has a substantial impact on convergence time
- Feedback metric: L is the length of the expression and V its value, $\max \frac{V}{(1.5)^L}$



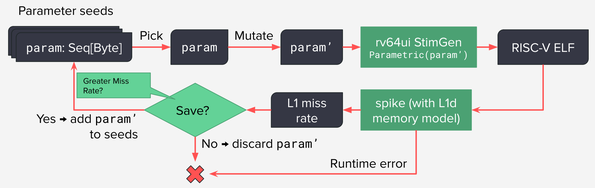
Sequence-Based CPU Instruction Generators

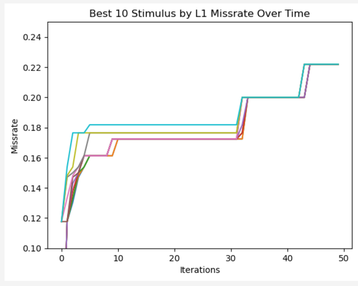


- Stimulus made up of sequences
- Sequences manually written to target uArch features
- Sequences query knobs to make random decisions

RV64UI StimGen Fuzzing

- A rv64ui stimulus generator with basic sequences (Inst, RAW, Loop, Branch) was created using this API
- The generator is put in a parametric fuzzing loop with spike's memory model





Conclusion and Future Directions

We demonstrate that a parametric stimulus generator with instrumentation enables effective hardware fuzzing.

Future Work

- Comparison against AFL-style fuzzers
- Feature complete rv64 instruction generator
- RTL coverage-driven feedback

https://github.com/girantinas/randomapi/

{rohaga, vighnesh.iyer, bora}@berkeley.edu