# VGF: Value-Guided Fuzzing
# Fuzzing Hardware as Hardware

Ruochen Dai[†], Michael Lee[†], Patrick Hoey[‡], Weimin Fu[*], Yier Jin[¶], Xiaolong Guo[*], Shuo Wang[†], Dean Sullivan[§], Tuba Yavuz[†], Orlando Arias[‡]

[†] University of Florida
[‡] University of Massachusetts Lowell
[*] Kansas State University
[¶] University of Science and Technology of China
[§] University of New Hampshire

# Have less buggy hardware

Have less buggy hardware

Dynamically generate test vectors to check assertions.

# The Issue With Static Analysis

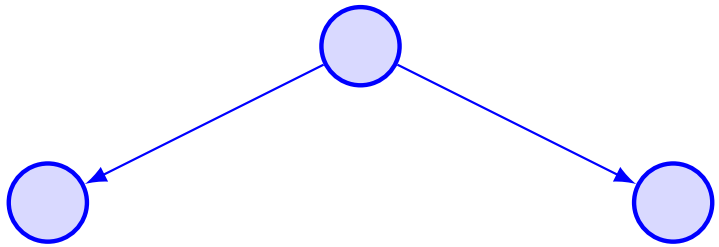
Explosion of states.

# The Issue With Static Analysis

Explosion of states.

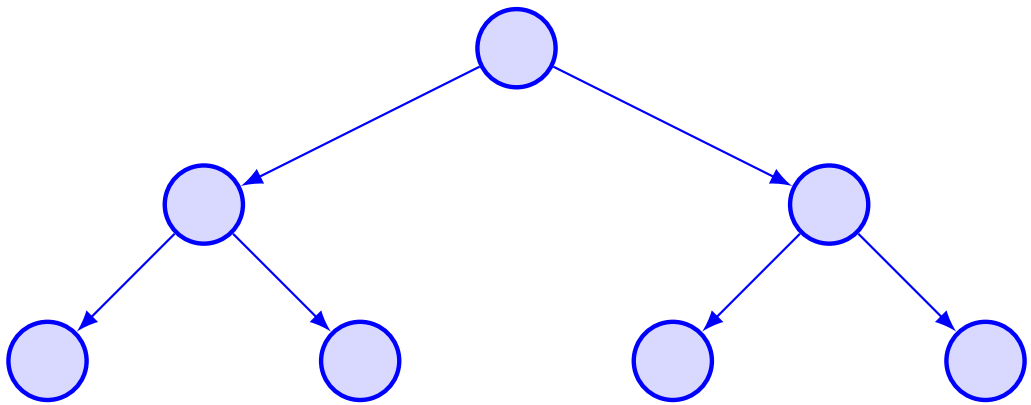
# The Issue With Static Analysis

Explosion of states.



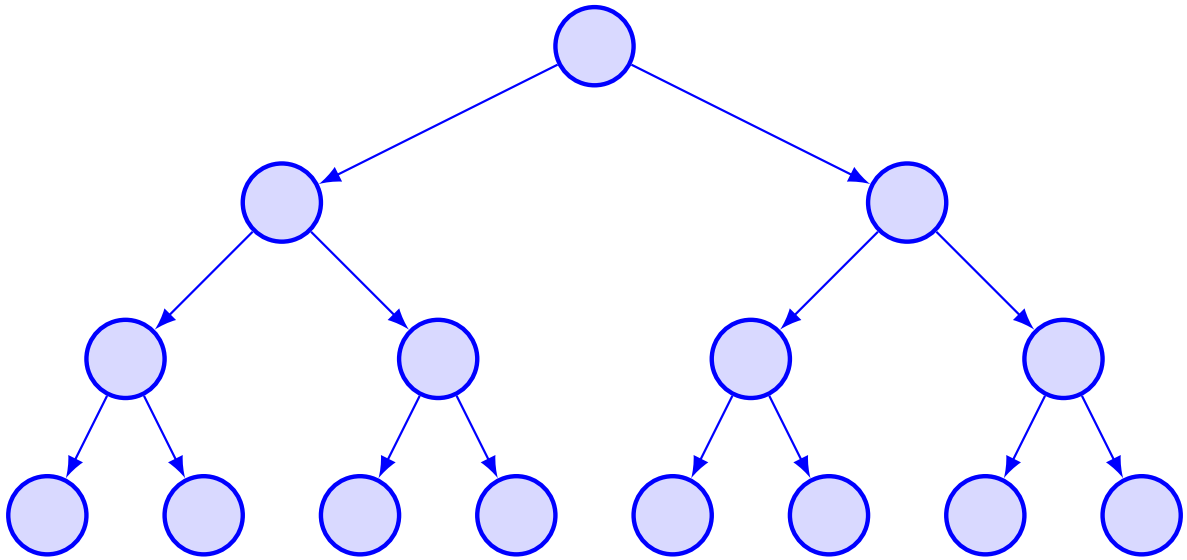
# The Issue With Static Analysis

Explosion of states.



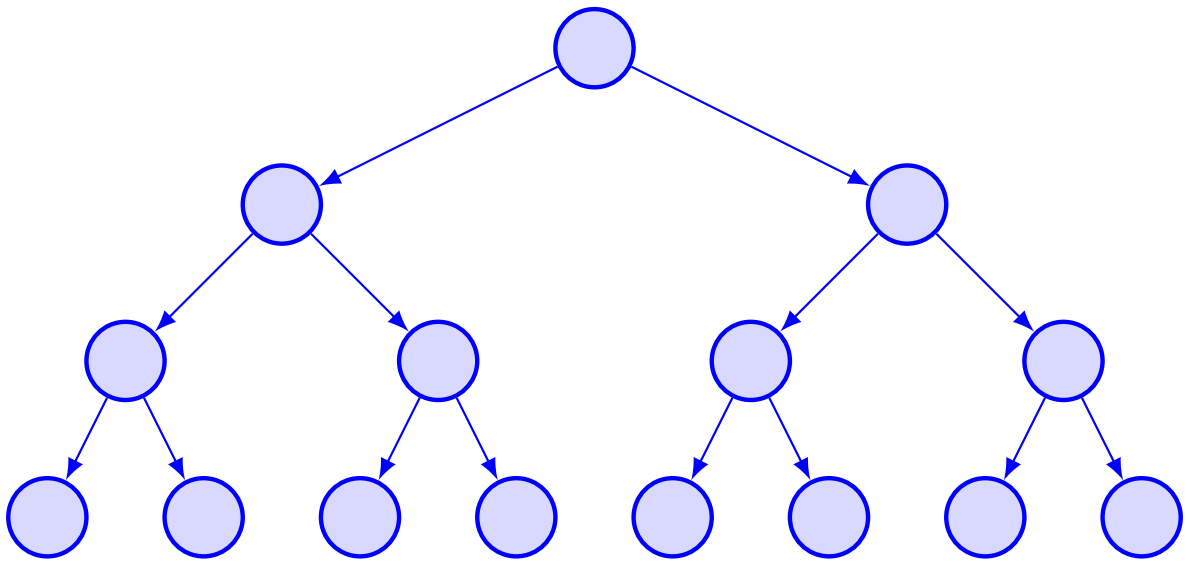
# The Issue With Static Analysis

Explosion of states.



# The Issue With Static Analysis

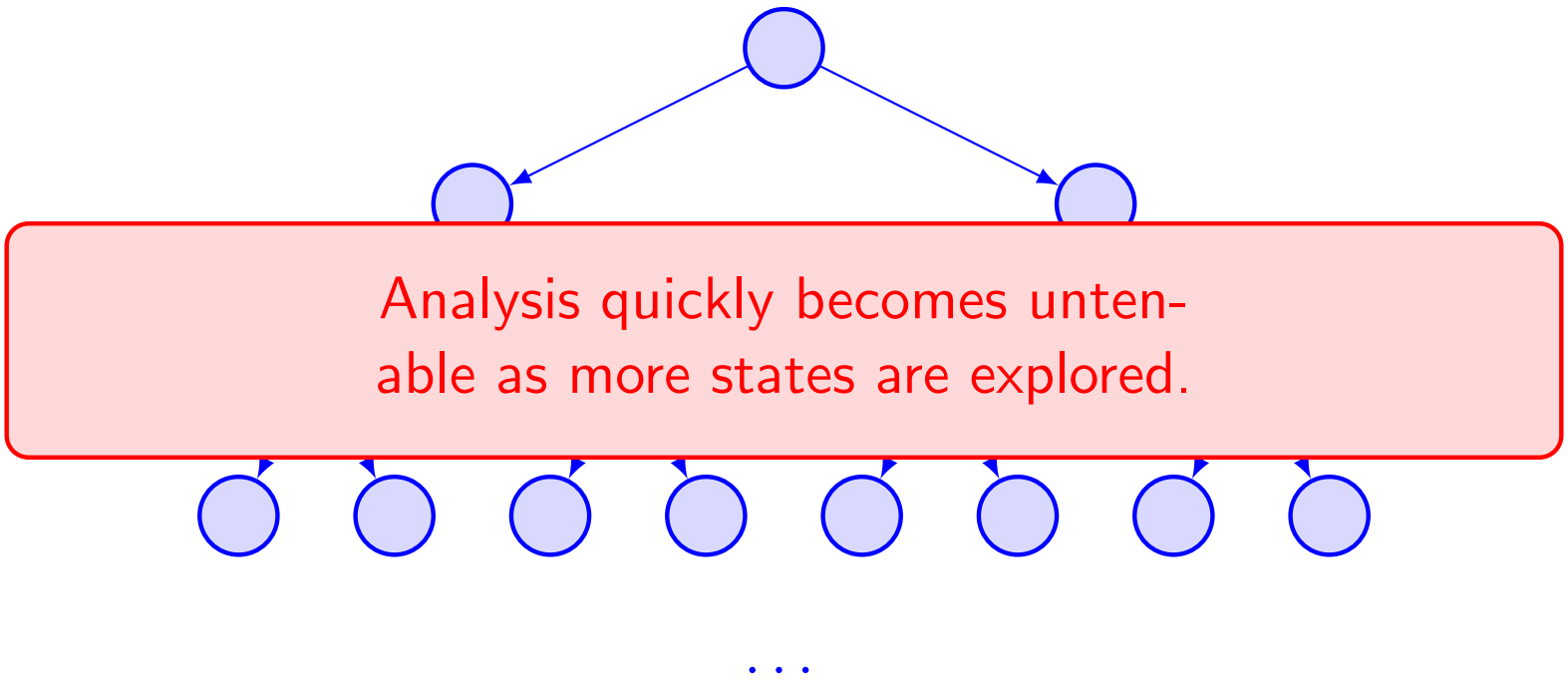
Explosion of states.



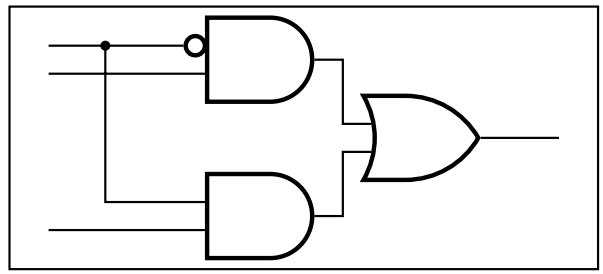
...

# The Issue With Static Analysis

Explosion of states.

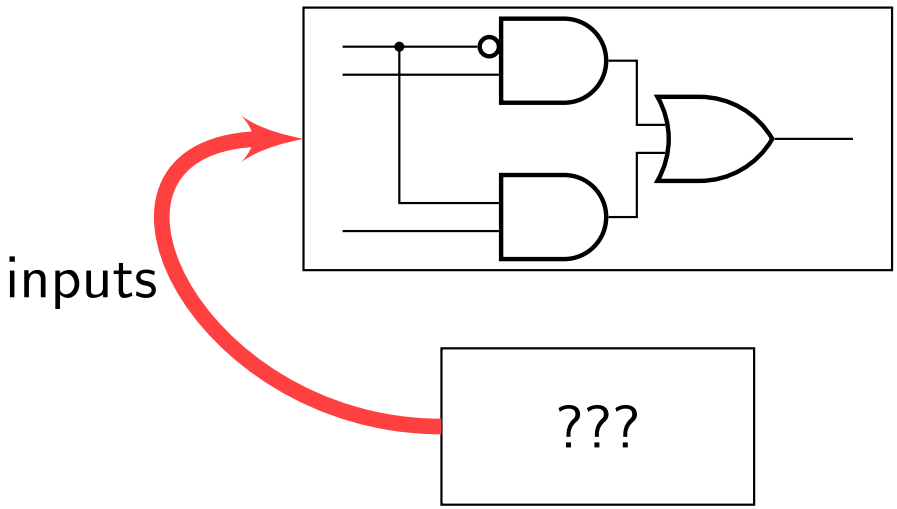


# Dynamic Analysis of Hardware?

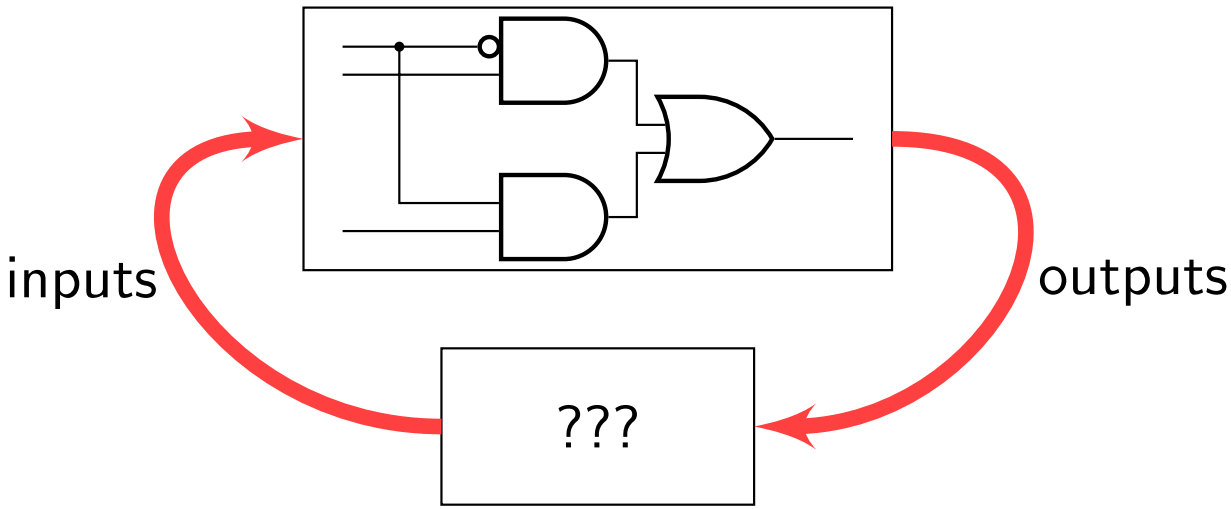


???

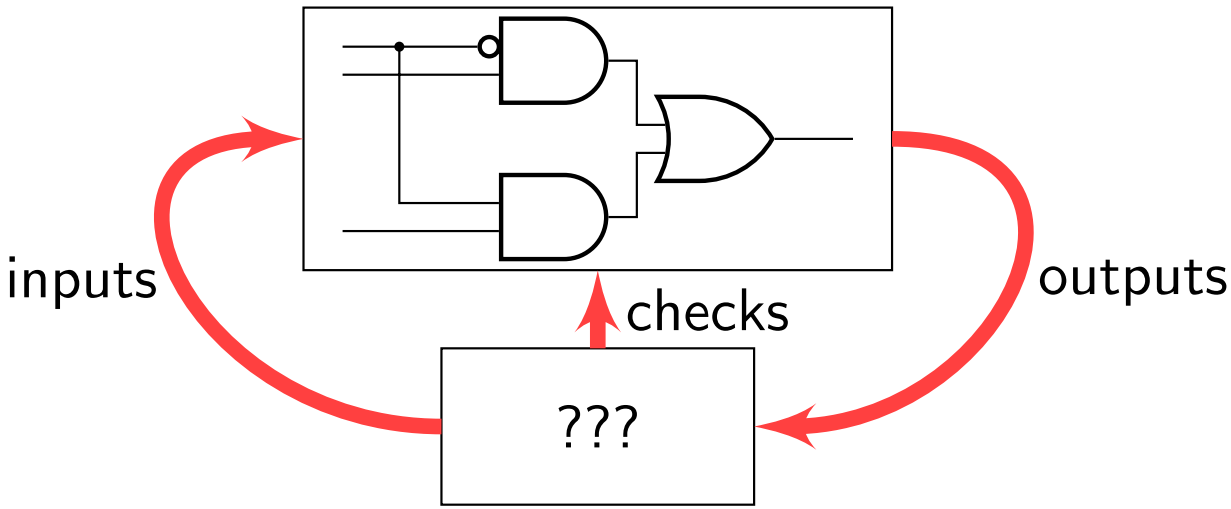
# Dynamic Analysis of Hardware?



# Dynamic Analysis of Hardware?



# Dynamic Analysis of Hardware?



# Fuzz Testing Hardware?

Previous work

- Convert hardware model into C/C++ model
- Instrument resulting code
- Use something like AFL to fuzz

# Fuzz Testing Hardware?

Previous work

- Convert hardware model into C/C++ model
- Instrument resulting code
- Use something like AFL to fuzz

## Survey Paper

W. Fu, O. Arias, Y. Jin, and X. Guo, "Fuzzing hardware: Faith or reality? : Invited paper," in *2021 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, 2021, pp. 1–6

# A Closer Look at Previous Work

Assume that conversion into C/C++ model is 100% accurate.
Use basic block as measure of coverage.
Use lines visited as measure of coverage.
Does the approach always work?

# A Closer Look at Previous Work

Assume that conversion into C/C++ model is 100% accurate.
Use basic block as measure of coverage.
Use lines visited as measure of coverage.
Does the approach always work?

## Implicit control-flow

```
module mux(output o,
           input a, b, s);
    assign o = s ? b : a;
endmodule
```

## No control-flow

```
module mux(output o,
           input a, b, s);
    assign o = |{a & ~s, b & s};
endmodule
```

# The Many Faces of a Multiplexer

```
module mux(output o,
           input a, b, c);
    assign o = (a & ~s) | (b & s);
endmodule
```

```
module mux(output logic o,
           input a, b, c);
    always_comb @(*)
        if(s) o = b;
        else o = a;
endmodule
```

```
module mux(output o,
           input a, b, c);
    assign o = |{a & ~s, b & s};
endmodule
```

```
module mux(output o,
           input a, b, c);
    assign o = s ? b : a;
endmodule
```

```
module mux(output o,
           input a, b, c);
    wire t[1:0];
    and u0(t[0], a,   ~s);
    and u1(t[1], b,    s);
    or  u2(o   , t[0], t[1]);
endmodule
```

```
primitive mux(o, a, b, s);
    output o;
    input a, b, s;
    table
        0   ?   0   : 0;
        1   ?   0   : 1;
        ?   0   1   : 0;
        ?   1   1   : 1;
        ?   ?   x   : x;
    endtable
endprimitive
```

```
module mux(output o,
           input a, b, c);
    wire s_inv;
    pmos u0(s_inv, 1, s);
    nmos u1(s_inv, 0, s);
    cmos u2(o, a, s_inv, s);
    cmos u3(o, b, s, s_inv);
endmodule
```

# The Many Faces of a Multiplexer

```
module mux(output o,
           input a, b, c);
    assign o = (a & ~s) | (b & s);
endmodule
```

```
module mux(output logic o,
           input a, b, c);
    always_comb @(*)
        if(s) o = b;
        else o = a;
endmodule
```

```
module mux(output o,
           input a, b, c);
    assign o = |{a & ~s, b & s};
endmodule
```

```
module mux(output o,
           input a, b, c);
    assign o = s ? b : a;
endmodule
```

```
module mux(output o,
           input a, b, c);
    wire t[1:0];
    and u0(t[0], a,    ~s);
    and u1(t[1], b,     s);
    or  u2(o   , t[0], t[1]);
endmodule
```

```
primitive mux(o, a, b, s);
    output o;
    input a, b, s;
    table
        0   ?   0   : 0;
        1   ?   0   : 1;
        ?   0   1   : 0;
        ?   1   1   : 1;
        ?   ?   x   : x;
    endtable
endprimitive
```

```
module mux(output o,
           input a, b, c);
    wire s_inv;
    pmos u0(s_inv, 1, s);
    nmos u1(s_inv, 0, s);
    cmos u2(o, a, s_inv, s);
    cmos u3(o, b, s, s_inv);
endmodule
```

# The Many Faces of a Multiplexer

```
module mux(output o,
           input a, b, c);
    assign o = (a & ~s) | (b & s);
endmodule
```

```
module mux(output logic o,
           input a, b, c);
    always_comb @(*)
        if(s) o = b;
        else o = a;
endmodule
```

```
module mux(output o,
           input a, b, c);
    assign o = |{a & ~s, b & s};
endmodule
```

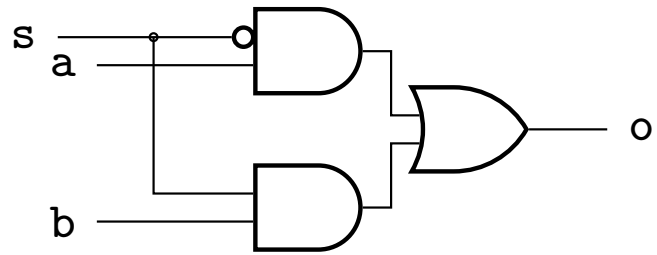
```
module mux(output o,
           input a, b, c);
    assign o = s ? b : a;
endmodule
```

```
module mux(output o,
           input a, b, c);
    wire t[1:0];
    and u0(t[0], a,   ~s);
    and u1(t[1], b,    s);
    or  u2(o   , t[0], t[1]);
endmodule
```

```
primitive mux(o, a, b, s);
    output o;
    input a, b, s;
    table
        0   ?   0   : 0;
        1   ?   0   : 1;
        ?   0   1   : 0;
        ?   1   1   : 1;
        ?   ?   x   : x;
    endtable
endprimitive
```

```
module mux(output o,
           input a, b, c);
    wire s_inv;
    pmos u0(s_inv, 1, s);
    nmos u1(s_inv, 0, s);
    cmos u2(o, a, s_inv, s);
    cmos u3(o, b, s, s_inv);
endmodule
```

# Design Coverage?



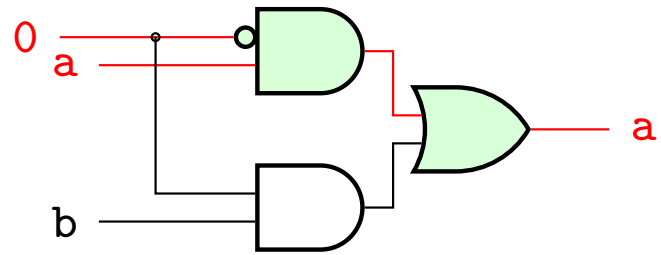

## Gate Level Circuit

```
module mux(output o,
           input a, b, s);
    wire t[1:0];
    and u0(t[0], a,   ~s);
    and u1(t[1], b,    s);
    or  u2(o   , t[0], t[1]);
endmodule
```

## Equivalent High-Level Circuit

```
module mux(output logic o,
           input a, b, s);
    always_comb @(*)
        if(s) o = b;
        else o = a;
endmodule
```

# Design Coverage?



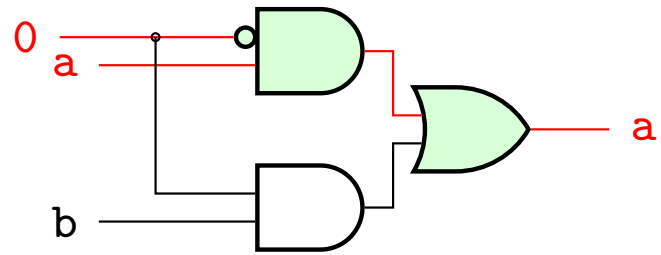

## Gate Level Circuit

```
module mux(output o,
           input a, b, s);
    wire t[1:0];
    and u0(t[0], a,   ~s);
    and u1(t[1], b,    s);
    or  u2(o   , t[0], t[1]);
endmodule
```

## Equivalent High-Level Circuit

```
module mux(output logic o,
           input a, b, s);
    always_comb @(*)
        if(s) o = b;
        else o = a;
endmodule
```

# Design Coverage?



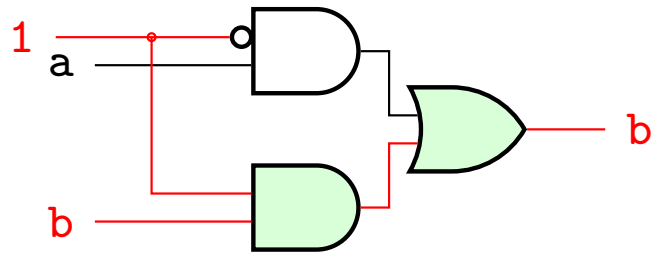

## Gate Level Circuit

```
module mux(output o,
           input a, b, s);
    wire t[1:0];
    and u0(t[0], a,   ~s);
    and u1(t[1], b,    s);
    or  u2(o   , t[0], t[1]);
endmodule
```

## Equivalent High-Level Circuit

```
module mux(output logic o,
           input a, b, s);
    always_comb @(*)
        if(s) o = b;
        else o = a;
endmodule
```

# Design Coverage?



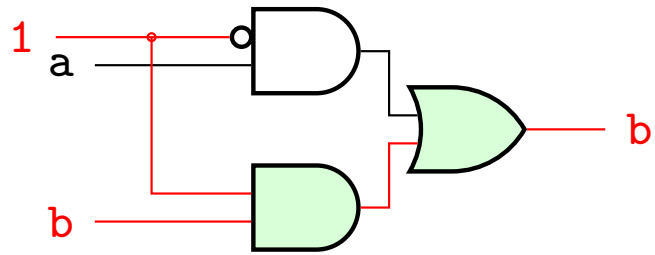

## Gate Level Circuit

```
module mux(output o,
           input a, b, s);
    wire t[1:0];
    and u0(t[0], a,   ~s);
    and u1(t[1], b,    s);
    or  u2(o   , t[0], t[1]);
endmodule
```

## Equivalent High-Level Circuit

```
module mux(output logic o,
           input a, b, s);
    always_comb @(*)
        if(s) o = b;
        else o = a;
endmodule
```

# Design Coverage?




## Gate Level Circuit

```
module mux(output o,
           input a, b, s);
    wire t[1:0];
    and u0(t[0], a,   ~s);
    and u1(t[1], b,    s);
    or  u2(o   , t[0], t[1]);
endmodule
```

## Equivalent High-Level Circuit

```
module mux(output logic o,
           input a, b, s);
    always_comb @(*)
        if(s) o = b;
        else o = a;
endmodule
```

# Looking Closer at Coverage

## Gate Level Circuit

```
module mux(output o,
           input a, b, s);
    wire t[1:0];
    and u0(t[0], a,   ~s);
    and u1(t[1], b,    s);
    or  u2(o   , t[0], t[1]);
endmodule
```

## Equivalent High-Level Circuit

```
module mux(output logic o,
           input a, b, s);
    always_comb @(*)
        if(s) o = b;
        else o = a;
endmodule
```

## Observations

- Gate level circuit has 100% line coverage on a single simulation tick
- Gate level circuit has no control-flow information

# Verilated Output?

Looking at the C++ conversion of each HDL style:

## Gate Level Circuit

```
vlSelfRef.o = (((~ (IData)(vlSelfRef.s)) & (IData)(vlSelfRef.a))
        | ((IData)(vlSelfRef.b) & (IData)(vlSelfRef.s)));
```

## Equivalent High-Level Circuit

```
vlSelfRef.o = ((IData)(vlSelfRef.s)
        ?
            (IData)(vlSelfRef.b)
        :   (IData)(vlSelfRef.a));
```

# The Insight

Instead of looking as lines of code or control-flow information, look at the state space of the design.

# Tracking Signal Values

Can we obtain the values of signals at simulation time?

# Tracking Signal Values

Can we obtain the values of signals at simulation time?

- **Yes!**
- Most HDL simulators implement the Verilog Procedural Interface
- Use VPI to interface with and extend simulator!

# Tracking Signal Values

Can we obtain the values of signals at simulation time?

- **Yes!**
- Most HDL simulators implement the Verilog Procedural Interface
- Use VPI to interface with and extend simulator!
- Have a design-agnostic VPI harness, launch simulator with harness

# Tracking Signal Values

Can we obtain the values of signals at simulation time?

- **Yes!**
- Most HDL simulators implement the Verilog Procedural Interface
- Use VPI to interface with and extend simulator!
- Have a design-agnostic VPI harness, launch simulator with harness
- Integrate harness with a fuzzer for input generation

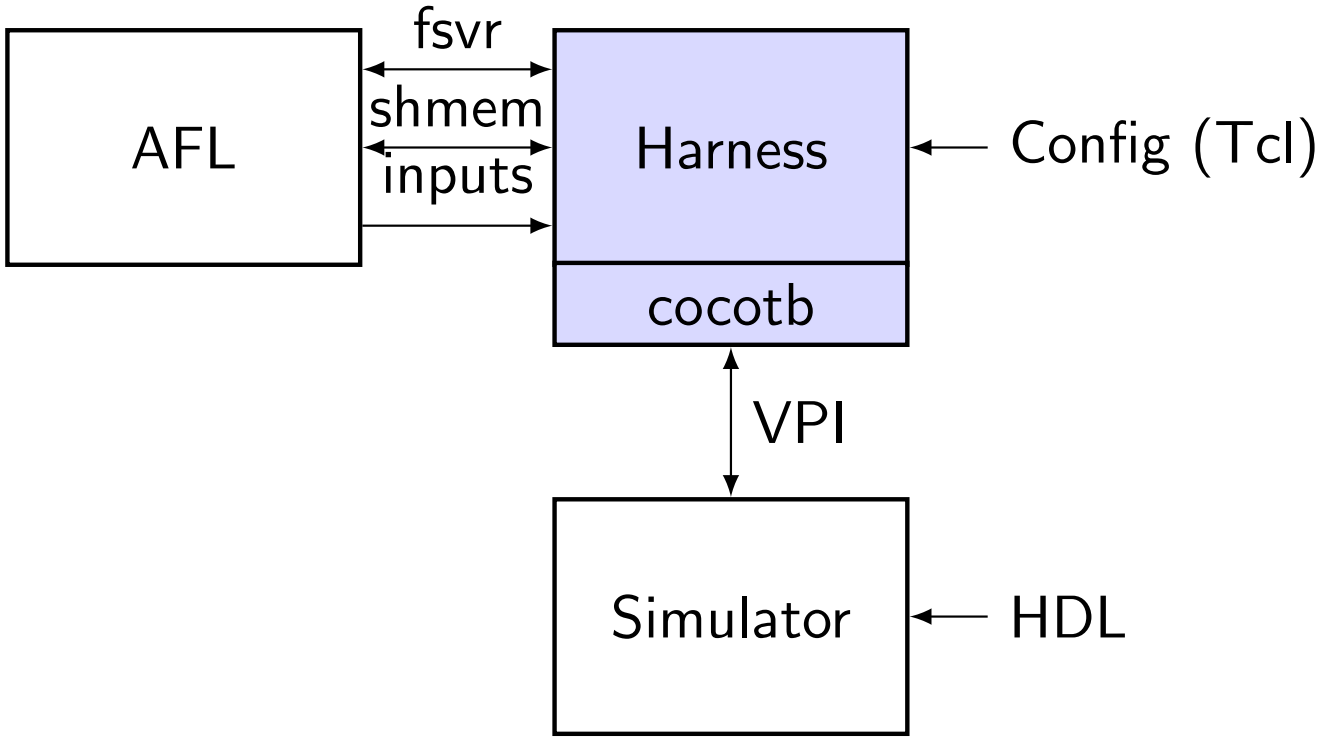
# VGF Methodology

- Tell harness what assertions to check for
- Tell harness which signals to track
- Tie harness to a fuzzing engine
- Use signal changes to provide feedback to the fuzzer

# VGF Methodology

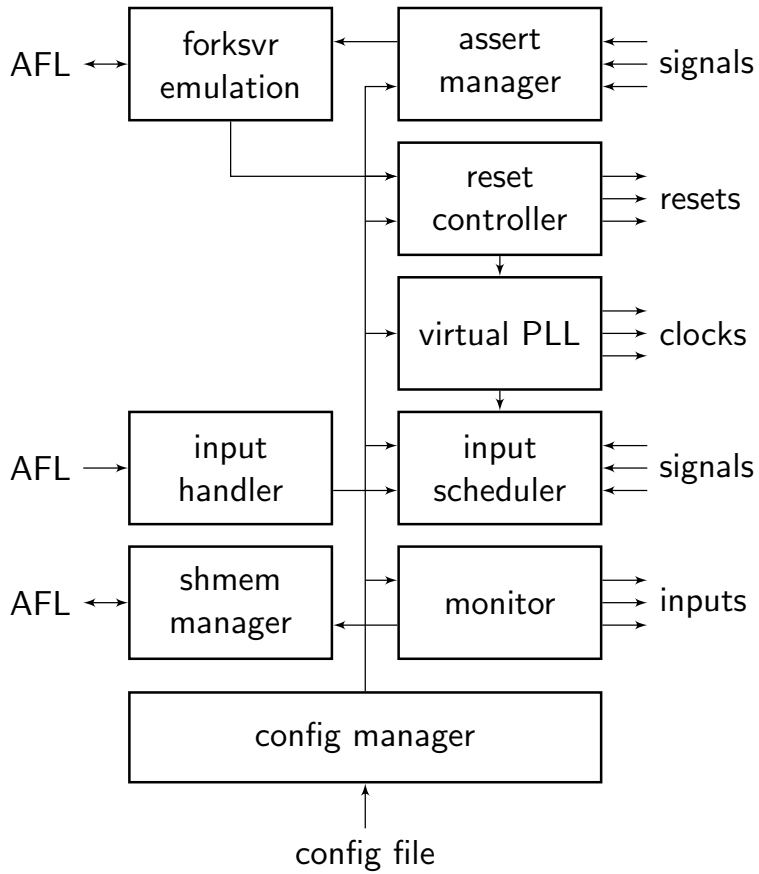
- Tell harness what assertions to check for
- Tell harness which signals to track
- Tie harness to a fuzzing engine
- Use signal changes to provide feedback to the fuzzer
- Let the system run

# VGF Architecture



# Harness Architecture

- Use VPI to communicate with DUT
- Communication with AFL done through UNIX pipes and shared memory
- Harness configuration done using Tcl script



# The AFL Instrumentation

```
if(condition) {
    current_location = <COMPILER_RANDOM_NUMBER_0>;
    shared_memory[current_location ^ previous_location]++;
    previous_location = current_location >> 1;
    /* ... */
} else {
    current_location = <COMPILER_RANDOM_NUMBER_1>;
    shared_memory[current_location ^ previous_location]++;
    previous_location = current_location >> 1;
    /* ... */
}
```

# The AFL Instrumentation

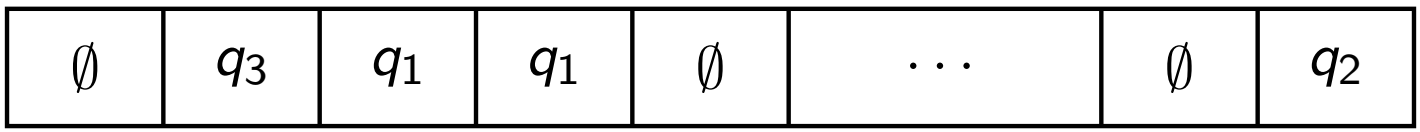
- 64 KiB allocation as `shared_memory`
  - Name exported in environment variable `__AFL_SHM_ID`
- Shift done to avoid issues with loops:

```
do {
    current_location = <COMPILER_RANDOM_NUMBER_2>;
    shared_memory[current_location ^ previous_location]++;
    previous_location = current_location >> 1;
    /* code with no branches or loops */
} while(condition);
```

if not using shift, `0 == current_location ^ previous_location`

# AFL Deciding on Input

For every bucket in the control-flow hash map, AFL keeps a *preferred* input queue. The input queue contains a copy of the data fed as input to the fuzzed program.




If a more favorable input is found for a particular bucket, AFL replaces the queue for that bucket with the more favorable one.

# AFL Reward Function

- AFL prioritizes inputs that are small and yield fast executions
- For a given input queue q, AFL computes

  ```
  fav_score = q->exec_us * q->len
  ```

  and tries to minimize this number.
- Number of buckets is not used on this reward

# AFL Reward Function

- AFL prioritizes inputs that are small and yield fast executions
- For a given input queue q, AFL computes

  ```
  fav_score = q->exec_us * q->len
  ```

  and tries to minimize this number.
- Number of buckets is not used on this reward
- More triggers that need to be resolved by the simulator imply that execution is slower!
  - Recall that we are using simulators that actually obey timing things like `#10 assign a = b;`

# AFL Reward Function

- AFL prioritizes inputs that are small and yield fast executions
- For a given input queue q, AFL computes

  ```
  fav_score = q->exec_us * q->len
  ```

  and tries to minimize this number.
- Number of buckets is not used on this reward
- More triggers that need to be resolved by the simulator imply that execution is slower!
  - Recall that we are using simulators that actually obey timing things like `#10 assign a = b;`
- Reward function is unsuitable for our purposes!

# New Reward Functions

New reward functions have been added to AFL.

- Geometric mean: $\sqrt[n]{\prod^{n} b_k}$
- Arithmetic mean: $\frac{1}{n} \sum^{n} b_k$
- Total value: $\sum^{n} b_k$
- Total buckets: $n$

where $n$ is the number of non-zero buckets in the hash map and $b_k$ the value in a non-zero bucket. We try to maximize these values.

# VGF vs. Formal Verification

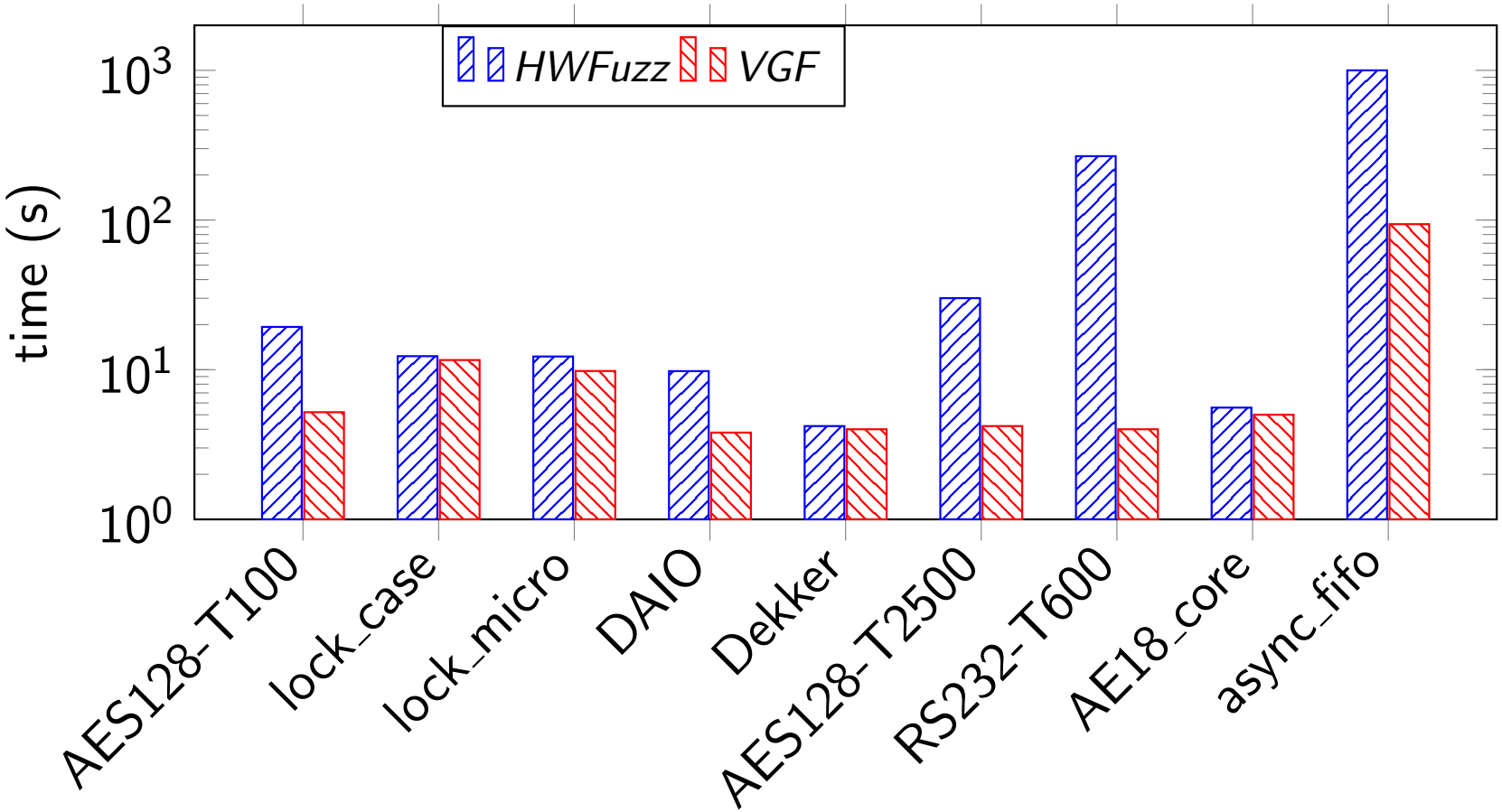
| Design | VGF | EBMC | FPV |
|---|---|---|---|
| AES128-T100 | 5.2 | –† | 20.09 |
| lock_case | 11.6 | 0.11 | 18.41 |
| lock_micro | 9.8 | 422.29 | 17.16 |
| DAIO | 3.8 | 1.03 | 17.37 |
| Dekker | 4 | 0.18 | 17.25 |
| Unidec | –† | –‡ | 17.92 |
| AES128-T2500 | 4.2 | –† | 25.04 |
| RS232-T600 | 4 | 0.31 | 17.35 |
| AE18_core | 5 | 10.66 | 17.11 |
| async_fifo | 93.8 | –‡ | 17.37 |

† Allotted time limit reached, timeout

‡ Tool encountered a parsing error

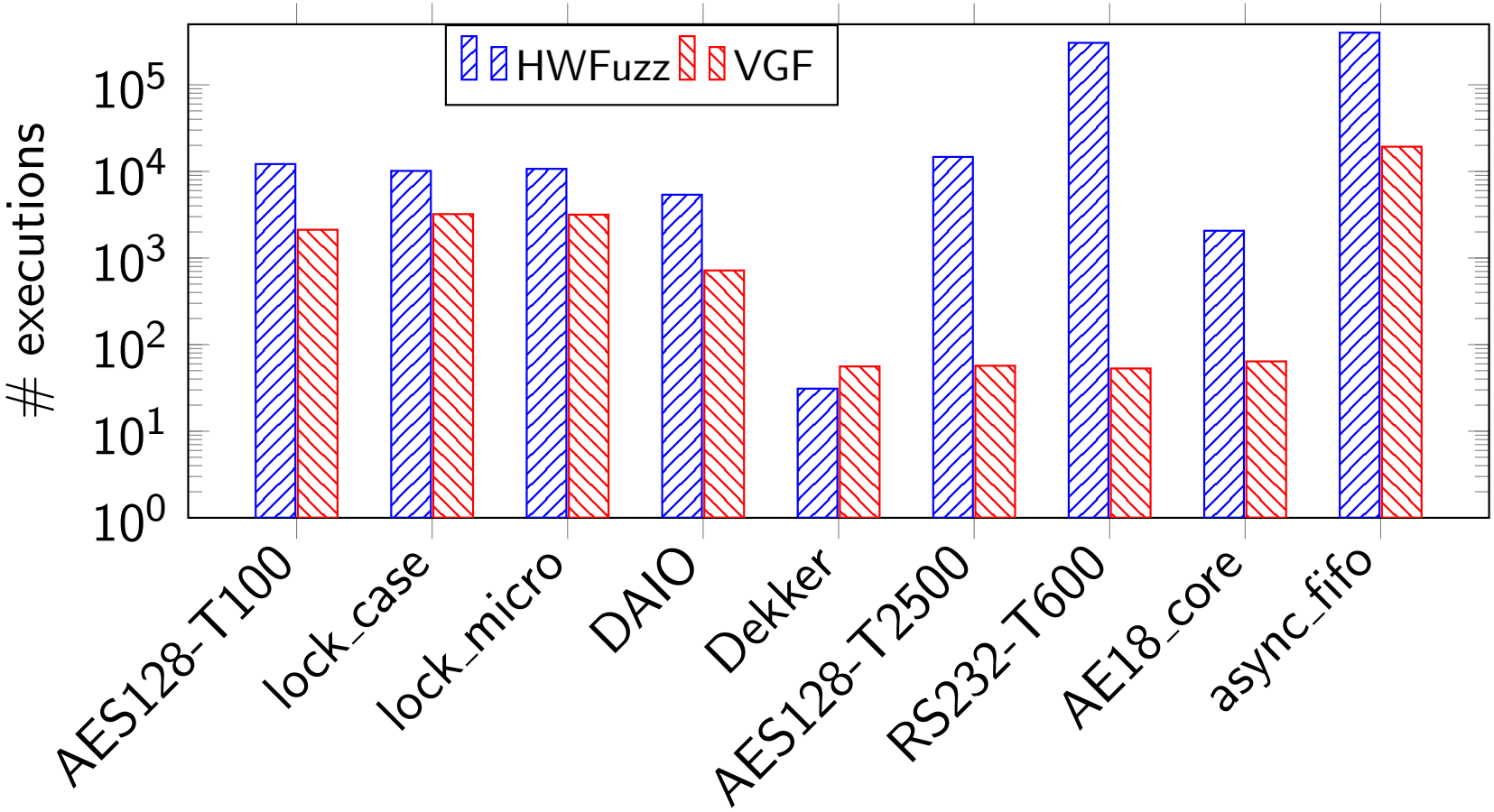
Table: Time to assertion trigger (in s) of VGF compared to Enhanced Bounded Model Checker (EBMC), and Synopsys VC Formal FPV.

# VGF vs. Hardware Fuzzing




T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3237–3254

# VGF vs. Hardware Fuzzing




T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3237–3254

# Conclusions

- It is possible to build a design agnostic *and* language agnostic simulation harness
- It is possible to trap hardware bugs with dynamic test vector generation
- We are faster than both open source fuzzers as well as formal verification tools

# Conclusions

- It is possible to build a design agnostic *and* language agnostic simulation harness
- It is possible to trap hardware bugs with dynamic test vector generation
- We are faster than both open source fuzzers as well as formal verification tools
- More testing is needed, larger designs are welcomed

# Questions?

orlando_arias@uml.edu