Reverse Engineering Heterogeneous Hardware Controller Designs via Finite State Machine Extraction for Improved Precision Taint Tracking

Rena Feng
Advisor: Professor Sharad Malik

Junior Independent Work Spring 2025
Department of Electrical and Computer Engineering
Princeton University

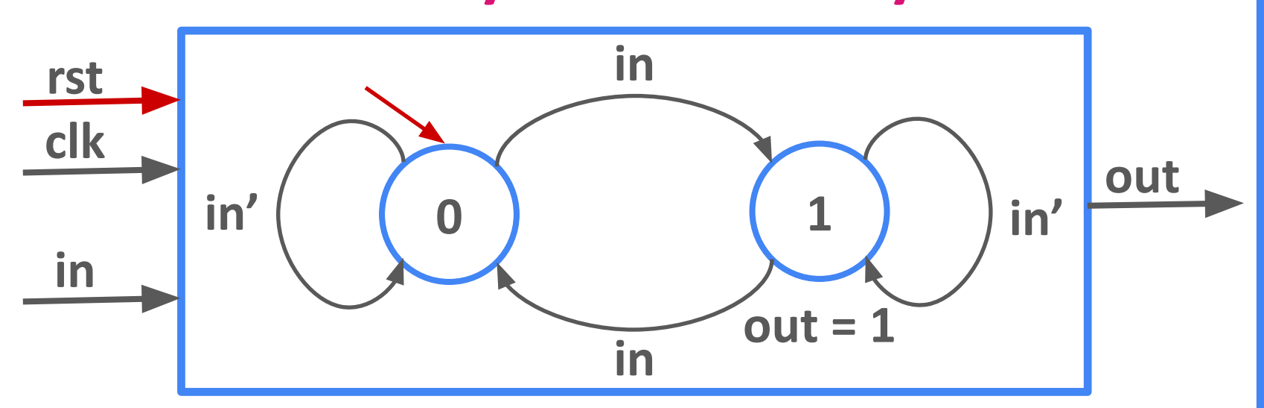
Abstract

Timing side-channel attacks exploit varying-length secret-dependent control flow paths in hardware circuits. While taint tracking with taint-kill can detect such leaks, it requires a control flow graph or finite state machine (FSM) that a Hardware Description Language (HDL) controller's syntax doesn't necessarily expose. This paper proposes methods to construct that graph from a simulation-derived truth table, enabling reliable taint-kill analysis on any style of HDL controller implementation.

Introduction and Background

A hardware circuit's timing can leak secret information, as attackers can infer internal states and values from execution duration. Information-flow analyses, like taint tracking with taint-kill [1], can determine whether a controller's timing depends on secret variables. However, this method requires a control flow graph that HDL modules (written in behavioral, structural, or mixed styles) do not always present transparently as FSMs. Simple text parsing fails to handle such diverse implementation styles. This project instead extracts each module's truth table via simulation and reverse engineers the corresponding control flow graphs. The paper details this style-agnostic process across various HDL implementations.

Structural vs Behavioral Controllers: How can a control flow graph or FSM be extracted from HDL implementations of different syntaxes and styles?



Structural Controller Body Code

```
reg  state;   // current state bit: 0=S0, 1=S1
wire next_state;    // next state bit

assign out  = state; // output logic

// next state logic
assign next_state = (state & ~in) | (~state & in);

always @(posedge clk) begin // state register
  if (rst)
    state <= 1'b0;  // reset to S0
  else
    state <= next_state;  // update to next state
end
```

Behavioral Controller Body Code

```
localparam STATE_S0 = 1'b0;  // S0
localparam STATE_S1 = 1'b1;  // S1
reg state;       // current state
reg next_state;  // next state

always @(*) begin // output logic
  out = 1'b0;
  case (state)
    STATE_S0: out = 1'b0;  // in S0, out=0
    STATE_S1: out = 1'b1;  // in S1, out=1
  endcase
end

always @(*) begin // next state logic
  next_state = state;
  case (state)
    STATE_S0: begin
      if (in) next_state = STATE_S1;
      else     next_state = STATE_S0;
    end
    STATE_S1: begin
       if (in) next_state = STATE_S0;
       else     next_state = STATE_S1;
    end
  endcase
end

always @(posedge clk) begin // state register
  if (rst)
    state <= STATE_S0;    // reset to S0
  else
    state <= next_state;  // update to next state
end
```

Methods

Brute Force Testbench

Inputs:
The controller under test, with its state space defined by the number of bits used to represent state.
All possible input values to be iterated over.

Output:
A .vcd waveform file capturing the simulation of every reset state and input combination.

Algorithm:

```
1. Iterate through all reset states: for each controller state that can be represented:
   a. Drive the controller into that reset state.
   b. For every combination of input values:
      i. Apply reset and pulse the clock once.
      ii. Release reset and pulse the clock again.
2. Finish: all possible states and input combinations have been tested via a brute force method.
```

Reachable States Only Testbench

Inputs:
The controller under test, with its state space defined by the number of bits used to represent state.
All possible input values to be iterated over.

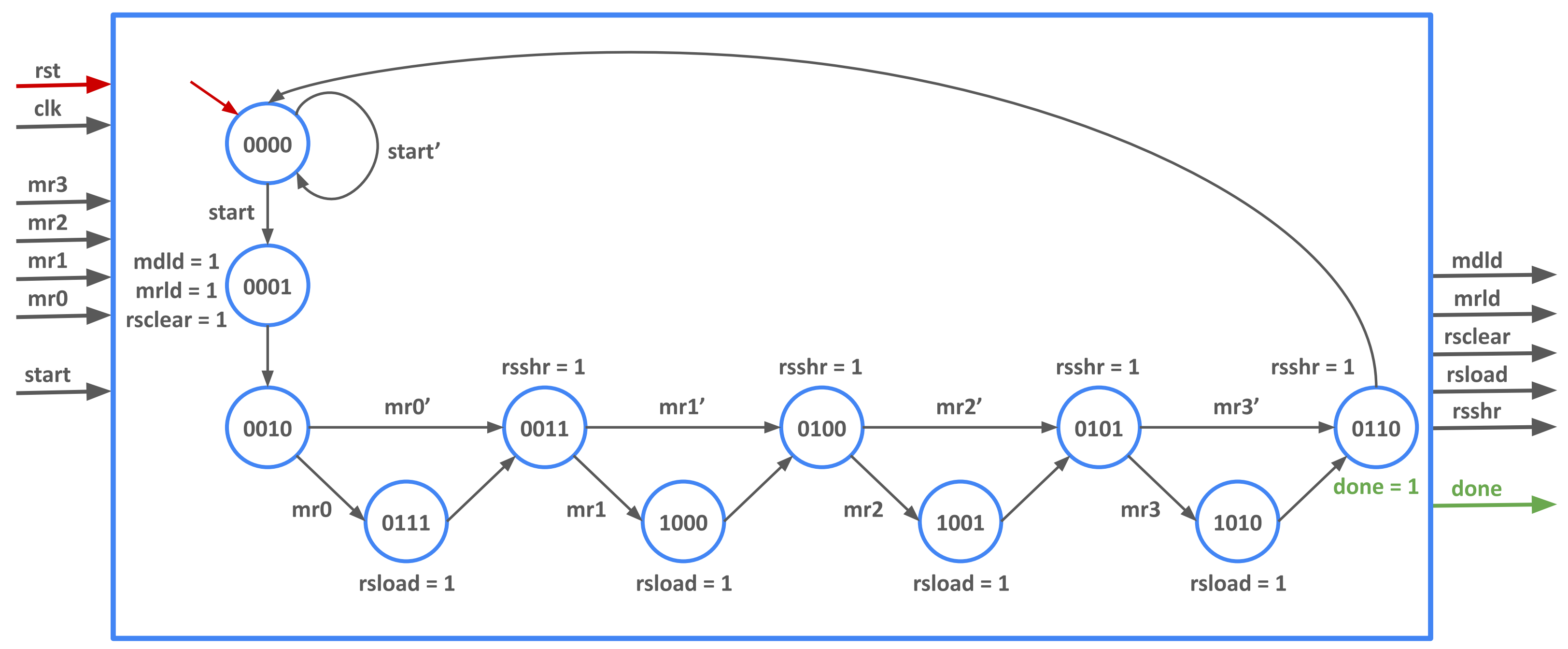
Output:
A .vcd waveform file capturing the simulation of every reset state and input combination.

Algorithm:

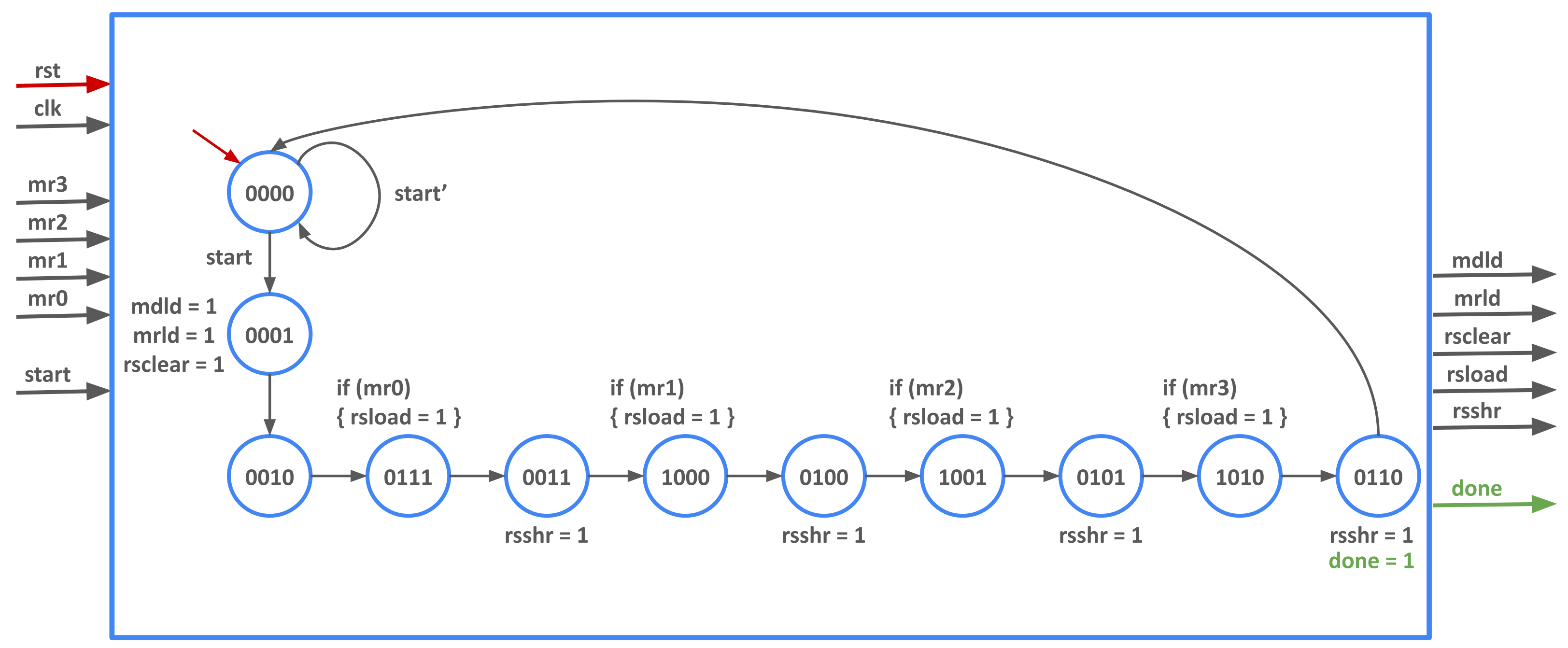
```
1. Begin by queuing only the reset state (state 0) for testing.
2. Explore queued states.
3. Repeat until all states in queue are explored:
   a. Pick the next state from the queue.
   b. For every combination of input values:
      i. Reset the controller into the picked state.
      ii. Apply the inputs.
      iii. Pulse the clock twice (once with reset asserted, once released).
      iv. Observe which state the controller goes to next.
      v. If that next state hasn't been seen before, add it to the queue.
      vi. Mark the picked state as "tested."
4. Finished: once every queued state is visited and no new states appear, the set of "tested" states is exactly the set of all reachable states.
```

Methods

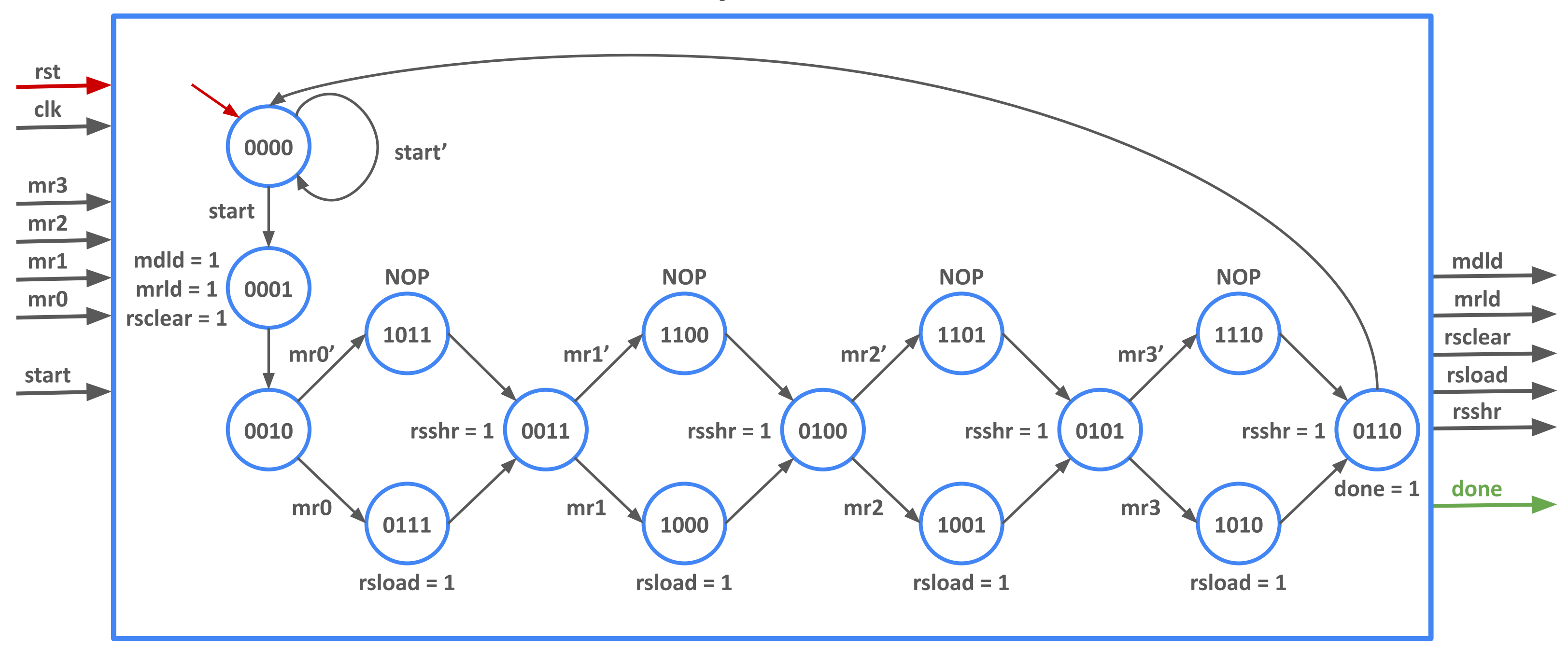
Controller 1: Violates Constant Time



Controller 2: Upholds Constant Time



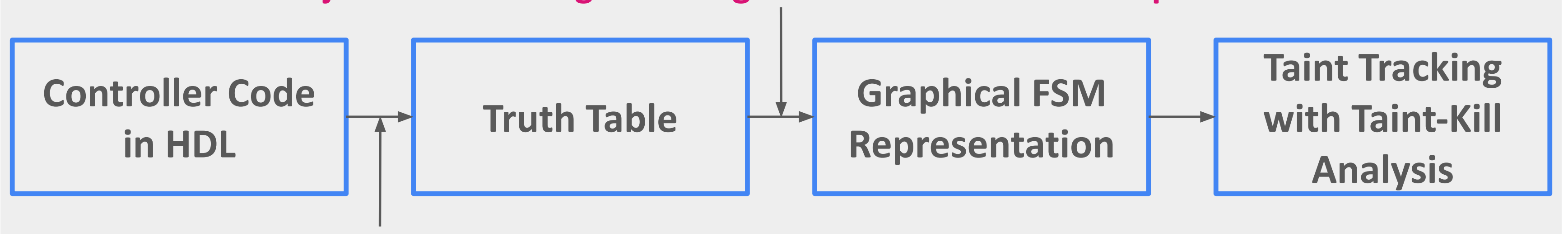
Controller 3: Upholds Constant Time



1. Three different controller finite state machines (FSMs) for a 4-bit shift multiplier [2], proposed by Vahid [2] and Marina [1], were implemented in both behavioral and structural Verilog.
2. Two Verilog testbenches extract state transition information from the controllers:
   a. "Brute Force": cycles through every possible combination of inputs and current states to enumerate all next state transitions
   b. "Reachable-Only": exercises only those state-input pairs reachable from the previous states
3. Simulation outputs → concise state transition truth table with the specific input vectors responsible for each transition (Python scripts)
4. Truth tables → .json files representing the FSMs of each controller (Python scripts)

Timing side-channel analysis of an HDL controller using taint tracking with taint-kill requires an FSM representation of its control flow.

Conversion to .json file showing each single state transition and dependent variables


Controller Code in HDL
Truth Table
Graphical FSM Representation
Taint Tracking with Taint-Kill Analysis

Verilog testbenches to extract truth table
(Brute Force and Reachable-Only simulation)
Post-processing of .vcd simulation output files

Results (4-bit Shift Multiplier Controller)

Controller 1: Taint Tracking with Taint-Kill

Input (.json FSM Graph Representation):

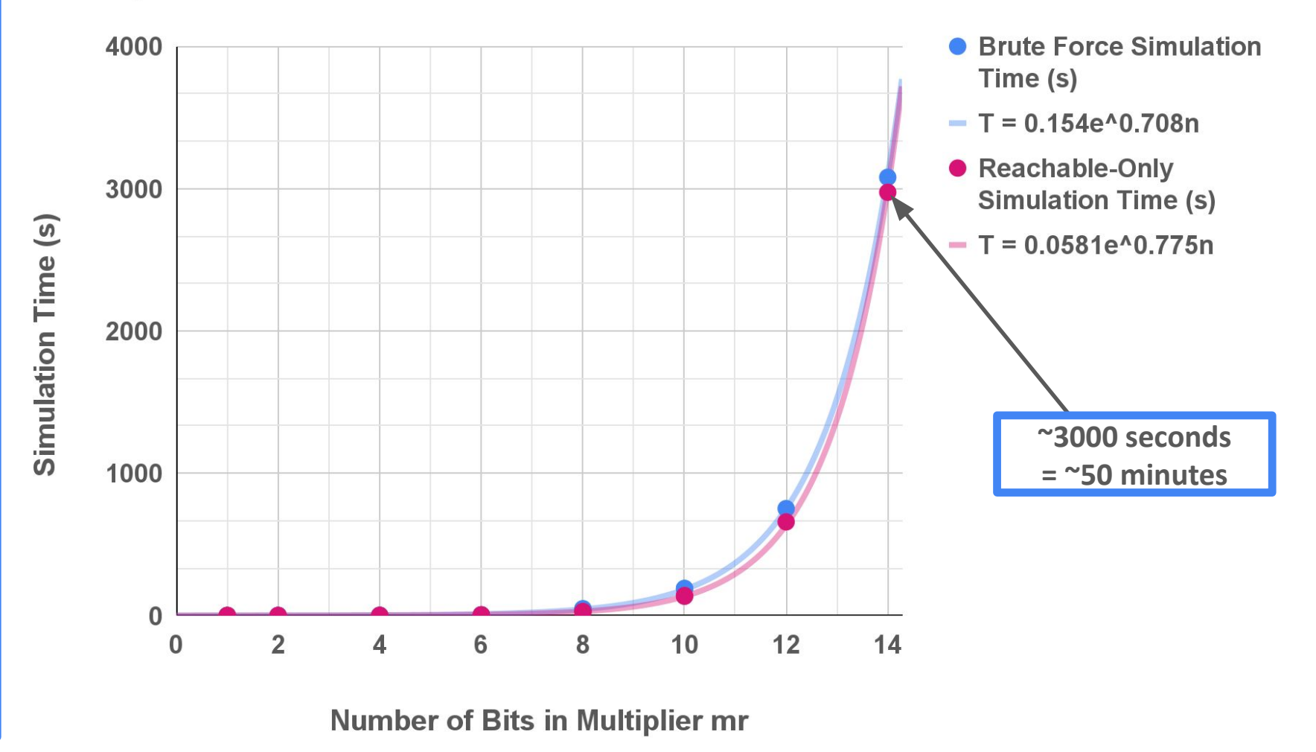
```
{'0': [['0', '1'], ['SMControlTest.control.start']],
 '1': [['10'], []],
 '10': [['11', '111'], ['SMControlTest.control.mr0']],
 '100': [['101', '1001'], ['SMControlTest.control.mr2']],
 '1000': [['100'], []],
 '1001': [['101'], []],
 '101': [['110', '1010'], ['SMControlTest.control.mr3']],
 '1010': [['110'], []],
 '11': [['100', '1000'], ['SMControlTest.control.mr1']],
 '110': [['0'], []],
 '111': [['11'], []]}
```

Output:

```
Dominator Nodes of Controller 1:
['0', '1', '10', '11', '100', '101', '110']

There is NO secret dependency (NO timing side-channel
vulnerability) in Controller 1: False
```

Controller 1: Simulation Time vs Number of Bits in Multiplier



Controller 2: Taint Tracking with Taint-Kill

Input (.json FSM Graph Representation):

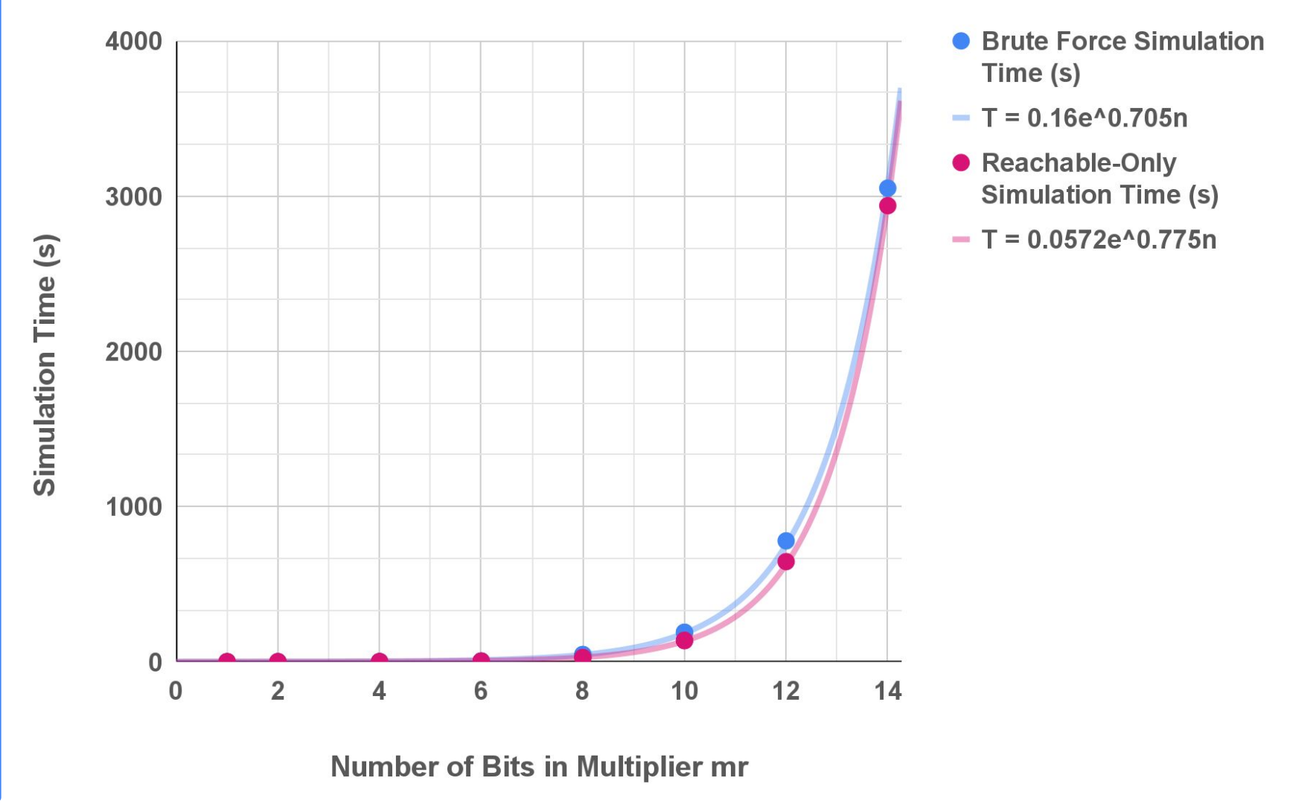
```
{'0': [['0', '1'], ['SMControlTest.control.start']],
 '1': [['10'], []],
 '10': [['111'], []],
 '100': [['1001'], []],
 '1000': [['100'], []],
 '1001': [['101'], []],
 '101': [['1010'], []],
 '1010': [['110'], []],
 '11': [['1000'], []],
 '110': [['0'], []],
 '111': [['11'], []]}
```

Output:

```
Dominator Nodes of Controller 2:
['0', '1', '10', '111', '11', '1000', '100', '1001',
'101', '1010', '110']

There is NO secret dependency (NO timing side-channel
vulnerability) in Controller 2: True
```

Controller 2: Simulation Time vs Number of Bits in Multiplier



Controller 3: Taint Tracking with Taint-Kill

Input (.json FSM Graph Representation):

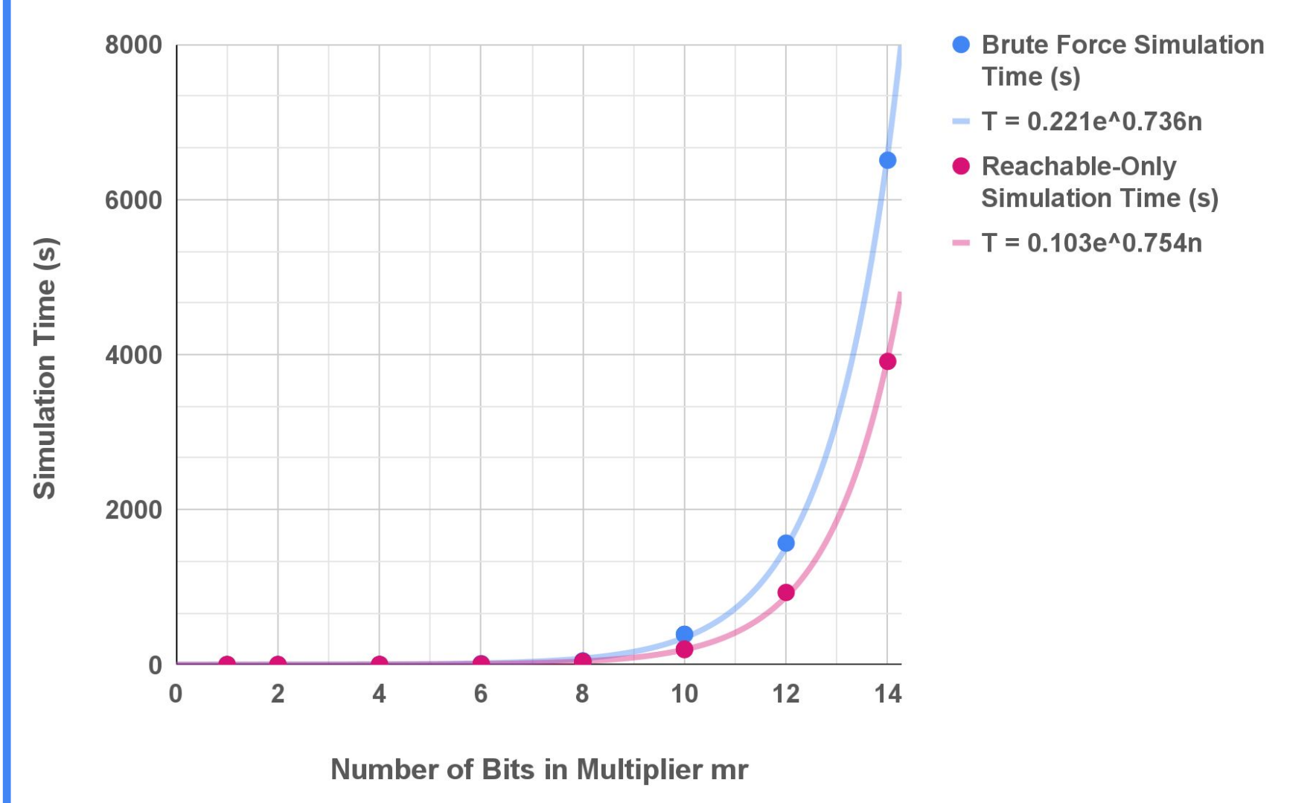
```
{'0': [['0', '1'], ['SMControlTest.control.start']],
 '1': [['10'], []],
 '10': [['1011', '111'], ['SMControlTest.control.mr0']],
 '100': [['1101', '1001'], ['SMControlTest.control.mr2']],
 '1000': [['100'], []],
 '1001': [['101'], []],
 '101': [['1110', '1010'], ['SMControlTest.control.mr3']],
 '1010': [['110'], []],
 '1011': [['11'], []],
 '11': [['1100', '1000'], ['SMControlTest.control.mr1']],
 '110': [['0'], []],
 '1100': [['100'], []],
 '1101': [['101'], []],
 '111': [['11'], []],
 '1110': [['110'], []]}
```

Output:

```
Dominator Nodes of Controller 3:
  ['0', '1', '10', '11', '100', '101', '110']

There is NO secret dependency (NO timing side-channel
vulnerability) in Controller 3: True
```

Controller 3: Simulation Time vs Number of Bits in Multiplier



Left:
★ Post testbench processing output .json file of FSM of the controller
★ Output of taint tracking with taint-kill algorithm applied to the controller

Right:
★ Graph of scalability of testbench (timing experiments)
  ■ Both Brute Force and Reachable-Only grow exponentially
  ■ 14-bit *mr*: ~50 minutes (Controllers 1 and 2) and ~ 110 minutes (Controller 3)

Conclusion and Discussions

As the timing of this algorithm scales exponentially with the number of bits used to represent states and transitions, optimizations in checking time are highly desirable to allow for further scalability:

★ Integrating graph analysis with the truth table extraction in the Verilog testbench
★ Recognizing isomorphic subgraphs: once one instance of a repeated pattern is proven to be constant time, identical iterations can be skipped

This project presents a method reverse engineering a graphical FSM from simulation data of hardware controllers written in arbitrary HDL styles. Information-flow analysis, like taint tracking with taint-kill, can then be performed on the extracted FSMs. Further optimizations will increase scalability and make the method applicable to a broader range of use cases.

References

[1] Sofia Marina, "Improving Taint Tracking Precision for Side Channel Vulnerability Detection," Fall 2024.
[2] F. Vahid, *Digital Design with RTL Design, VHDL, and Verilog*, 2nd ed. Hoboken, NJ, USA: John Wiley & Sons, Mar. 9, 2010.