Spring 2025

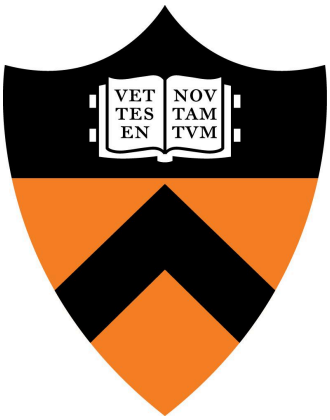
Junior Independent Work Report

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

Rena Feng

Advisor: Professor Sharad Malik (ECE)

Second Reader: Professor Aarti Gupta (COS)



Submitted in partial fulfillment

of the requirements for the degree of

Bachelor of Science in Engineering

Department of Electrical and Computer Engineering

Princeton University

I hereby declare that this Independent Work report represents my own work in accordance with University regulations.

I hereby declare that this Independent Work report does not include regulated human subjects research.

I hereby declare that this Independent Work report does not include regulated animal subjects research.

Rena Feng

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

Rena Feng
Advisor: Professor Sharad Malik (ECE)

Abstract

A hardware circuit's control flow may contain varying-length secret-dependent paths that can be exploited by timing side-channel attacks. Information flow analyses like taint tracking can detect whether a controller's timing depends on secret variables. With an additional "taint-kill" mechanism, the taint tracking algorithm is less conservative and gains accuracy. However, applying this method to a controller implemented in a Hardware Description Language (HDL) is difficult because syntactic and structural variations do not readily reveal the control flow graph, or finite state machine (FSM), required by the taint tracking algorithm. A text parser may not be able to extract the control flow from an unknown or unspecified format of controller code, but a testbench extracting the controller's truth table exposes its control flow behavior regardless of implementation style. This project proposes to build a control flow graph by extracting and processing the controller's truth table to enable taint tracking with taint-kill timing analysis on any HDL implementation.

Acknowledgements

Thank you to my advisor, Professor Sharad Malik, for your invaluable guidance, compassion, and encouragement throughout my first independent research project. Under your mentorship, I have gained my first real glimpse of hardware security and learned how to carry out a technical research project.

I am also grateful for the help of fellow student Sofia Marina (COS ’26), who graciously met with me to share insights about her independent work project from the previous semester, which served as a springboard for mine.

Thank you as well to Professor Aarti Gupta for serving as my second reader and to the ECE department for providing me with the opportunity and resources to explore independent research as an undergraduate.

Table of Contents

1. Introduction 5

2. Background and Related Works 6

2.1. Taint Tracking 6

2.2. Text Parsing 7

2.3. Hardware Description Language (HDL) Comparison: Chisel 8

2.4. Current Methods 9

3. Methods 10

3.1. Multiplier Controller Implementation 10

3.2. Reverse Engineering Truth Table for FSM Extraction 13

3.3. Optimized Truth Table Simulation Testbench 14

3.4. Post-Testbench Processing 15

3.5. Modified Taint-Kill Tracking Algorithm 17

4. Evaluation 17

4.1. Application of Taint Tracking 17

4.2. Simulation Timing 19

5. Future Improvements and Optimizations 21

6. Conclusions 22

References 23

Appendix 24

1. Introduction

Among the different ways an adversary (malicious party) can gain information about a system's private, or secret, data is through observing the amounts of time certain computations take. These attacks are called timing side-channel attacks. Hardware processes, whose operations may take different amounts of time depending on secret inputs, can set signals, which may not be secret themselves, that reveal these different inputs due to the different timing. If the time-dependent signal is measured, an adversary can thus learn about the inputs to the circuit.

One example is a multiplier with a "shortcut" where the output of the multiplication is immediately set to 0 if either of the inputs are 0. In all other cases, the full multiplication is carried out and takes the same time regardless of the inputs. If a *done* signal, which goes high to indicate when the computation completed, is accessible to an attacker, they can measure how much time elapses between setting the inputs and *done* changing to 1, allowing them to infer whether or not at least one of the inputs was 0.

Real world, larger scale timing side-channel attacks on hardware include Kocher's Timing Attack [1], where RSA key bits are inferred by measuring variations in the time required to perform private-key operations; Spectre, where sensitive data accessed via speculative execution is leaked by measuring cache timing [2]; and Meltdown, where out of order execution changes the cache contents, from which secrets can be inferred by comparing cache access times [3]. These attacks leak sensitive data and violate isolation guarantees between the user, systems, and/or processes. Such information leaks can be exploited by malicious actors to learn about users and/or processes.

A mitigation to prevent sensitive data from being leaked through timing side-channel attacks is to create constant time implementations of the processes. Constant time, in this case, means that the timing from when inputs are taken in by a module to when the outputs are produced is not dependent on the secret values. In many hardware circuit modules, a controller is used to dictate the control flow of the computation. If the control flow of the circuit is not dependent on secrets, then the circuit will not be susceptible to timing side-channel attacks.

Currently, there are ways (with varying degrees of accuracy) to verify whether a circuit's secret-dependent control flow operates with constant time. For example, one can run a taint tracking algorithm on the circuit's finite state machine (FSM) [4]. However, this control flow graph is not always easy to obtain from a circuit module coded in a style that may not directly map clearly onto an FSM.

This project begins by exploring controller implementation styles and hardware description languages (HDLs), attempting to discover if text parsing is a feasible method to extract an FSM structure from any implementation. It is ultimately determined that a strict text processing method would not suffice: in both the Verilog and Chisel HDLs, the code can be formatted in a variety of ways, such as behavioral, structural, or a mixture of both, and a clear format cannot be guaranteed. However, text parsing is discovered not to be necessary because a truth table can be extracted from any implementation of a module, from which a control flow graph can then be reverse engineered. This paper summarizes the process undertaken to explore characteristics across different implementations and HDLs and how their behavior can be extracted under a reverse engineering method that is independent of controller code style.

2. Background and Related Works

Previous attempts to accurately detect timing side-channel vulnerabilities inspire the methods in this paper as well as inform on the feasibility of approaches.

2.1. Taint Tracking

One of the formal verification methods for checking whether a controller has a secret-dependent timing path is taint tracking. Taint tracking works by marking secret inputs as "tainted" and propagating this taint through the circuit's control and data paths. If a tainted value influences control decisions, such as the state transitions of the FSM, the corresponding control paths are flagged as secret-dependent.

However, this technique is over-conservative and can result in false positives, where a circuit with no timing vulnerabilities is flagged for containing secret-dependent timing differences between control flow paths. For example, two secret-dependent control paths may reconverge after diverging and still complete in the same number of cycles. In such cases, even though the control sequence is influenced by secret data, there is no observable timing difference in the end. If the taint is not properly cleared when the paths merge, the analysis may incorrectly flag a timing leak where none exist [4].

A previous independent project by Sofia Marina introduced the idea of a “taint-kill” to eliminate the false positives by using breadth first search to find dominator nodes (points of reconvergence in the FSM) and removing the taint if all secret-dependent paths between two dominator nodes are of equal length (take the same number of clock cycles) [4].

The taint tracking with taint-kill method requires that the controller be represented in a graphical form in order to run the analysis, as it requires the presence of nodes and edges in order to find the dominator nodes and calculate path lengths. Thus, even with improved accuracy, there still remains a discrepancy between controller implementation and its graphical FSM representation that must be addressed in order to use the taint tracking with taint-kill timing verification method. How can an FSM graph be extracted from any controller code, that is, a controller implemented in any style in a given HDL?

2.2. Text Parsing

A text parsing algorithm was first considered as a possible means to convert controller code to an FSM graph. Glancing at behavioral implementation, this seems reasonable, as the code follows a pretty strict case and condition structure. Behavioral Verilog controllers often exhibit rigidly formatted current state output logic, next state logic, and register update sections that can be easily mapped onto an FSM. However, the style of programming cannot be guaranteed, especially since different designs may be easier to describe using a structural implementation or contain certain characteristics that necessitate a mix between structural and behavioral Verilog.

For instance, even though both Verilog controllers in Fig. 2 describe the FSM in Fig. 1, the structure and syntax of the code varies drastically. A text parser that expects a certain structure of code would fail to extract the FSM from the structural controller.

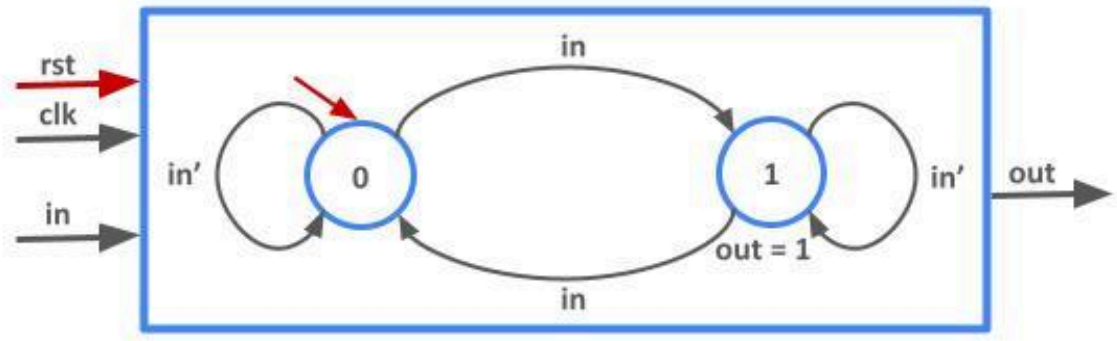


Figure 1: Simple controller FSM with two states

Structural Controller Body Code

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

// output logic
assign out = state;

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

// state register
always @(posedge clk) begin
  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

// output logic
always @(*) begin
  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

// next state logic
always @(*) begin
  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

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

Figure 2: Structural (left) vs behavioral (right) Verilog implementations of the FSM in Fig. 1

2.3. Hardware Description Language (HDL) Comparison: Chisel

An alternative hardware description language (HDL) was considered in hopes that its syntactic constraints would enable easier text parsing, as Verilog does not impose such formatting

restrictions. Chisel, an HDL growing in popularity for open source RISC-V processor implementations (used for the Rocket core [5]), was examined for distinct patterns that made the language inherently easier to visualize graphically than Verilog. If the language itself only allowed behavioral implementations, then the text parser would not have to account for various structural or mixed syntax implementations, in which case, it would be more feasible. However, it was discovered that Chisel was very similar to Verilog, and although it favored behavioral representations of hardware circuits, the coder could technically write in any form they wanted, which means that the graph extraction method should not rely on Chisel code being of a specific structure or style.

It is possible to get an intermediate representation of a circuit between Chisel and Verilog called FIRRTL (Flexible Intermediate Representation for RTL). One online tool, the Free Chips Project's Diagramming tool [6], can convert FIRRTL circuit representations into GraphViz graphs that visualize its netlist connections and specify physical linkages, such as a wire driving a port or an output feeding into a register. These graphs aren't able to immediately reveal control flow patterns and thus would not work for the taint tracking with taint-kill analysis. Additionally, a more general graph extraction method was desired due to the possibility of excluding many controller implementations by requiring a specific format in any language.

2.4. Current Methods

One method verifying constant time execution of hardware circuits is IODINE, which first transforms synthesizable Verilog into a clock-synchronous intermediate language called VINTER, where every process and assignment is made explicit on each clock tick, on which taint-based liveness checks are then performed [7]. While methods like these are promising, they include a lot of overhead in terms of handling and manipulating the code text. This project rejects text-based analysis of the controller code and moves towards a more encompassing method that works regardless of HDL or coding style. Instead of relying on fragile text parsing, simulation is employed via testbenches to extract truth tables, from which control flow graphs can be reverse engineered for taint tracking analysis.

3. Methods

3.1. Multiplier Controller Implementations

Often, hardware circuits employ modularity to promote reusability, simplify design, and enable scalable development through the composition of smaller, well-defined components. Therefore, the constant time verification process of such circuits can be simplified to verifying the constant time execution of each unique sub-component, or module, of the circuit. For example, if a 32-adder circuit consists of eight chained 4-bit adders, then the 32-bit adder exhibits secret independent constant time execution if the 4-bit adder module does.

Adders, multipliers, and other similar small, special purpose computational components, often come together to compose larger circuits such as processors or other systems whose sensitive information may be targeted by malicious actors. Thus, a small 4-bit shift multiplier, detailed in Frank Vahid's *Digital Design* (pages 375 - 377), was chosen to be the device under test for this project [8]. Three different controller FSMs for this multiplier were implemented in both behavioral and structural Verilog. Verilog was used as the HDL for this project, but as later revealed, the algorithm to extract the graphical FSM from the controller code is a general algorithm that could be implemented in any language to interface with any controller.

The multiplier, *mr*, is the secret input for this circuit, and the timing of focus is the time between when inputs are set and when the output *done* signal goes high (equals 1). For each bit of *mr*, if applicable, the running sum register is updated with the new sum via *rsload* and then shifted to the right by one bit via *rsshr* [8].

Controller 1: This controller has secret (*mr*) dependent paths as a "shortcut" will be taken to immediately shift if a bit of the multiplier *mr* is 0. The elapsed time from when the inputs are set to when *done* becomes 1 will, as a result, be inversely proportional to the number of 0s in the input *mr*.

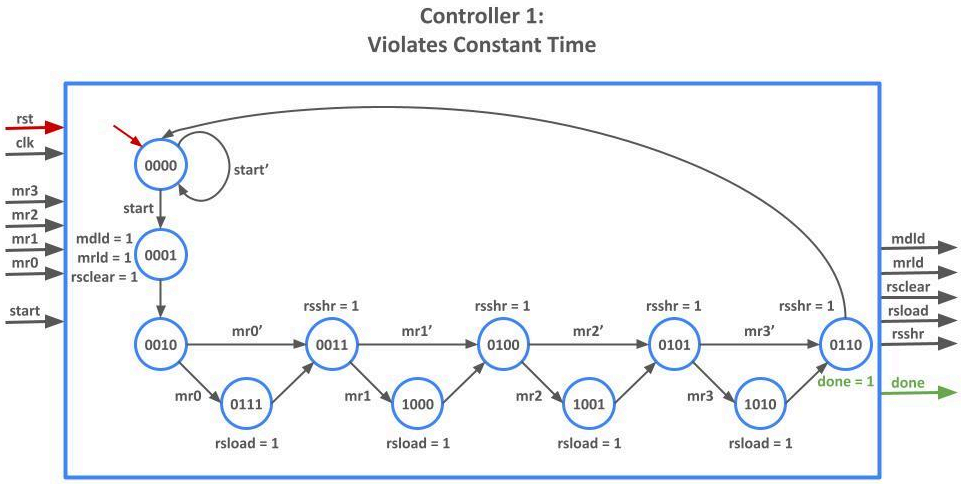


Figure 3: 4-bit Multiplier Controller 1: *rsload* set upon conditional visits to state [8]

Controller 2: This implementation of the control maintains the same time elapsed between all possible executions because the control flow transitions are independent of the value of *mr*. The *done* signal is set 10 cycles after the multiplication is signaled to start via the *start* signal no matter the value of *mr*. The condition for whether to set *rsload* to 1 turns into combinational logic within a state rather than the condition for a transition like in Controller 1.

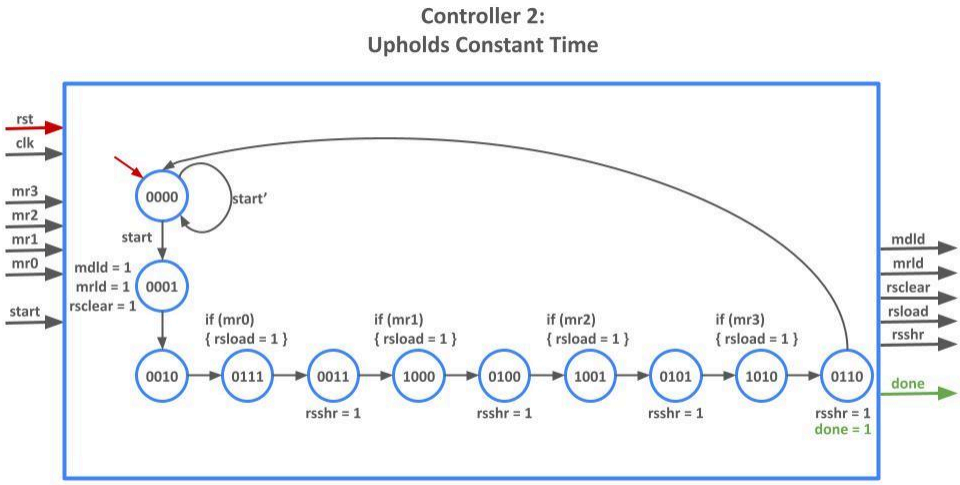


Figure 4: 4-bit Multiplier Controller 2: *rsload* set conditionally in an unconditionally-visited stat (modified version of Fig. 3 from Marina's "Improving Taint Tracking Precision" [4])

Controller 3: This implementation of the control maintains the same time elapsed between all possible executions because the control flow transitions, while dependent on the value of *mr*, do not change the length of the paths from the initial *start* to the *done* signal. In this case, although different values of *mr* cause visits to different states, the lengths of all possible *mr* dependent paths are the same: *done* is set 10 cycles after the multiplication is signaled to start via the *start* signal no matter the value of *mr*.

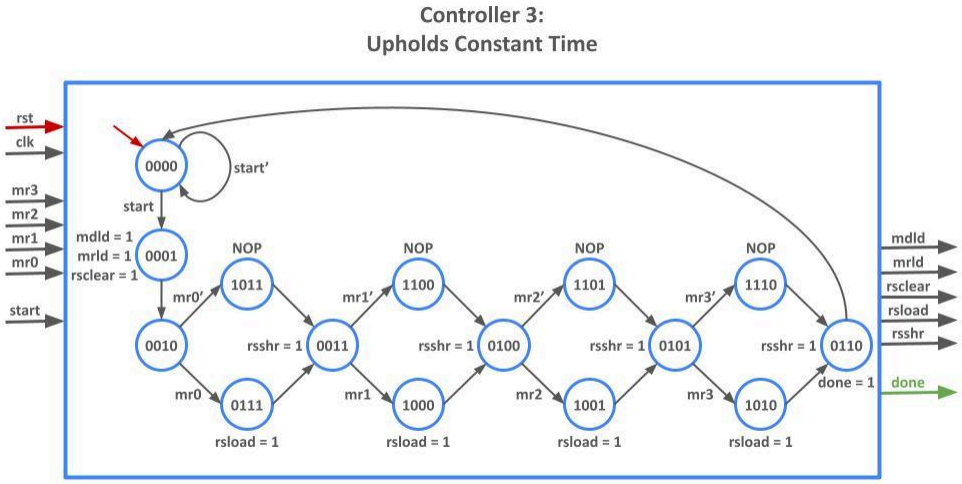


Figure 5: 4-bit Multiplier Controller 3: NOP states visited if *rsload* not updated
(Proposed by Marina in Fig. 4 of "Improving Taint Tracking Precision" [4])

The behavioral implementations of each controller were made by coding a 1-to-1 map of the graph of the FSM. The structural implementations were created by setting each bit of each output with the combination of the input bits that made each output bit high. The structural implementation required extracting a truth table from the graphical FSMs by encoding all of the input and output bit combinations.

This process of creating two drastically different implementations for the same logic for each of the three controllers further confirmed that a method beyond text parsing would be better suited for extracting the graphical FSM from a given code implementation. It was also discovered during this process that, as a truth table can be extracted from an FSM graph, an FSM graph can be constructed from a truth table.

Regardless of the syntactic style of a controller's code implementation, it has inputs and outputs that can be transcribed in a truth table. If a truth table can be obtained from an HDL controller implementation, then its graph representation can be constructed, and the taint tracking with taint-kill mechanism can be used to find timing vulnerabilities of the control. So, the problem has been reduced to extracting the truth table from any controller implementation.

3.2. Reverse Engineering Truth Table for FSM Extraction

For a Verilog controller like those implemented in Section 3.1., a testbench can be used to extract the truth table by cycling through all possible combinations of inputs and current states to get the transition. For each state, until all states in the state space are reached, every possible combination of inputs are tested to get every possible transition from that given state to a next state.

To allow for interfacing with such a testbench, the controller must undergo the following small modifications:

a) The reset condition must inject a desired state value, *reset_state*, upon reset. The *reset_state* value should be an input to the module so that the testbench can inject the base state from which to transition from. In other words, this *reset_state* will be sent from the testbench to the module as an input, and upon reset, the current state is set to the *reset_state*. To be able to do this, the number of bits used to encode the states (state space) must be known to the testbench as well.
b) The current state *s* needs to be made an output of the module so that transition mappings from *reset_state* to *s* can be visible in the testbench simulation results.

The simulation works by injecting a specific base state (*reset_state*) into the controller and then cycling through all possible combinations of inputs, always starting from that base state. By observing the resulting next state for each input combination and repeating this for every valid state representable by the number of state bits, the full transition behavior of the FSM is reconstructed. This testbench employs a "brute force" method as it exhaustively iterates through

all input combinations for every *reset_state* value representable by the state space, whether they are actually reachable states or not.

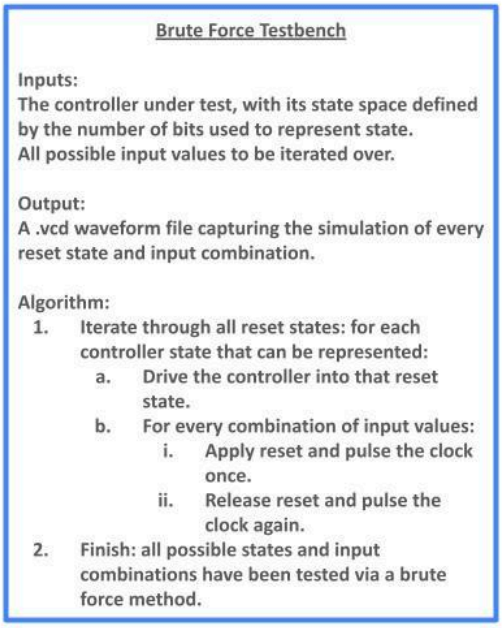


Figure 6: Brute force algorithm for simulating all possible state-to-state transitions in state space

3.3. Optimized Truth Table Simulation Testbench

The initial method performs a brute-force exploration of all possible input combinations. To optimize this process, one simplification involves restricting the analysis to reachable states only. Although the number of bits used to encode the state space defines a fixed range of potential state values, they are not necessarily all utilized by the FSM. By reducing the *reset_state* values to only those representing reachable states, unnecessary evaluations of unreachable transitions are avoided, thereby improving simulation efficiency.

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.

Figure 7: "Reachable-only" algorithm for simulating all possible reachable state-to-reachable state transitions

3.4. Post-Testbench Processing

Python was used to process the .vcd files from the simulation, extract the truth tables, simplify them, and turn them into the desired graph form, on which the taint tracking algorithm can then be applied (Fig. 8). The .json file generated by this process (Fig. 9) is the final product of the finite state machine extraction. It lists each current state and what the possible next states are from that state, as well as any variables that determine transitions to those next states. The output .json files for Controllers 1, 2, and 3 are shown in Fig. 10 and match the FSM designs presented in Figs. 3, 4, and 5.

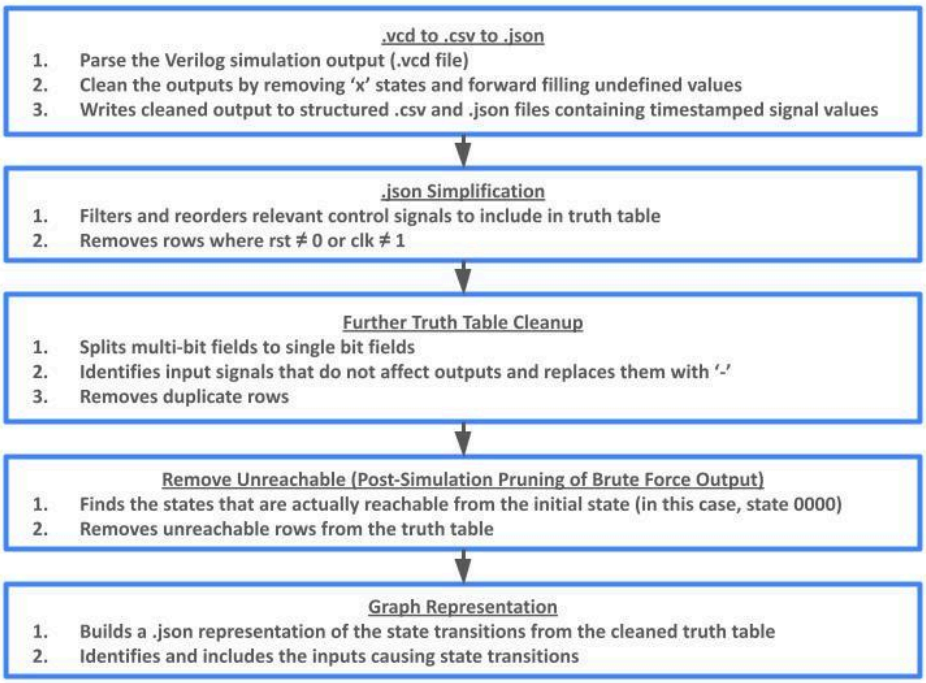


Figure 8: A series of Python functions transform the .vcd simulation output file to a .json representation of the controller's FSM, which can then be fed into the taint tracking with taint-kill algorithm for timing analysis

FSM Graph .json Format

```
{
    "<current_state>": [
        ["<next_state1>", "<next_state2>", …],
        ["<trigger_var1>", "<trigger_var2>", …]
    ],
    "<another_state>": [
        ["<next_stateX>", "<next_stateY>", …],
        ["<trigger_varA>", …]
    ],
    …
}
```

Figure 9: Format of output .json file of post-testbench processing represents FSM

Controller 1

```
{'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'], []]}
```

Controller 2

```
{'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'], []]}
```

Controller 3

```
{'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'], []]}
```

Figure 10: Output .json for Controllers 1, 2, and 3, respectively

3.5. Modified Taint Tracking Algorithm

Marina's original taint tracking with taint-kill code was adjusted to not revisit a state in a transition onto itself.. Her original example graphs did not include this corner case, but Controllers 1, 2, and 3, all exhibit this case in state 0000. Here, it is possible to stay in state 0000 forever if *start* always equals 0, and this can cause an infinite loop in the graph analysis. The updated algorithm was then used on the three .json files produced by the FSM extraction for Controllers 1, 2, and 3.

4. Evaluation

4.1. Application of Taint Tracking

The taint tracking with taint-kill algorithm correctly identifies the dominator nodes as well as whether the controllers have secret-dependent timing path differences. As seen in Fig. 11, the dominator nodes, or the states that all control flow paths visit, are correctly identified for each of the three controllers. Moreover, a secret dependency is detected for Controller 1 but not for Controllers 2 and 3, which is accurate (Fig. 12).

Using Taint Tracking with Taint-Kill Algorithms Developed & Written by Sofia Marina (COS '26)

Dominator Node Identification

Python Function Definition

```
""" finds which nodes must be on every path from START to END (these are denominator nodes) and returns
them in a list in BFS order
graph: .json graph representation
start: binary representation of node representing start state
end: binary representation of node representing end state """
def find_dominators(graph, start, end):
```

Calling the Function

```
print("Dominator Nodes of Controller 1: ")
print(find_dominators(graphController1, '0', '110'))
print("Dominator Nodes of Controller 2: ")
print(find_dominators(graphController2, '0', '110'))
print("Dominator Nodes of Controller 3: ")
print(find_dominators(graphController3, '0', '110'))
```

Output

```
Dominator Nodes of Controller 1:
['0', '1', '10', '11', '100', '101', '110']
Dominator Nodes of Controller 2:
['0', '1', '10', '111', '11', '1000', '100', '1001', '101', '1010', '110']
Dominator Nodes of Controller 3:
['0', '1', '10', '11', '100', '101', '110']
```

Figure 11: Dominator node analysis output for Controllers 1, 2, and 3

Using Taint Tracking with Taint-Kill Algorithms Developed & Written by Sofia Marina (COS '26)

Secret Dependency Identification

Python Function Definition

```
""" returns False if there is a pair of denominators where paths are unequal and there is a secret
dependency; returns True otherwise
graph: .json graph representation
start: binary representation of node representing start state
end: binary representation of node representing end state
secret_vars: list of secret variables """
def check_secret_dependency(graph, start, end, secret_vars):
```

Calling the Function

```
print("There is NO secret dependency (NO timing side channel vulnerability) in Controller 1: ")
print(check_secret_dependency(graphController1, '0', '110', secrets))
print("There is NO secret dependency (NO timing side channel vulnerability) in Controller 2: ")
print(check_secret_dependency(graphController2, '0', '110', secrets))
print("There is NO secret dependency (NO timing side channel vulnerability) in Controller 3: ")
print(check_secret_dependency(graphController3, '0', '110', secrets))
```

Output

```
There is NO secret dependency (NO timing side channel vulnerability) in Controller 1:
False
There is NO secret dependency (NO timing side channel vulnerability) in Controller 2:
True
There is NO secret dependency (NO timing side channel vulnerability) in Controller 3:
True
```

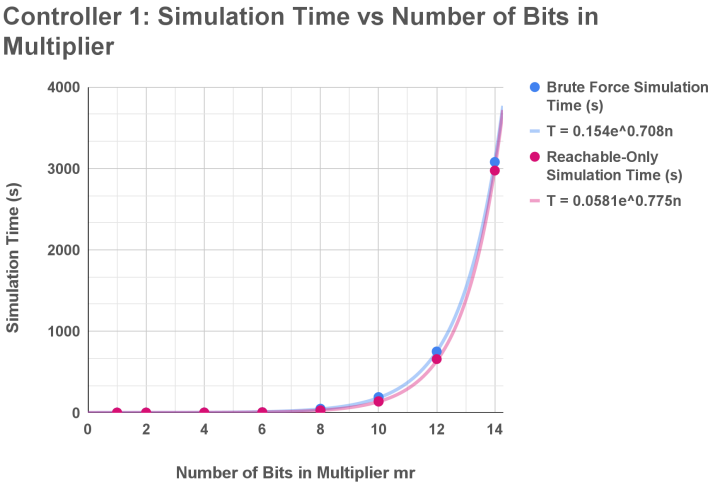
Figure 12: Secret dependency analysis output for Controllers 1, 2, and 3

4.2. Simulation Timing

The original multiplier controllers were implemented for a 4-bit multiplier, but additional behavioral controller implementations with a parameterized number of bits were created for timing and scalability analysis. These controllers are the same as Controllers 1, 2 and 3, with the only addition being that the FSM logic is expanded to encompass a variable number of bits in the multiplier, which the testbench can set.

As expected, the simulation time increases exponentially with the number of bits in the multiplier. This is because the number of bits is directly proportional to the number of FSM states, and a larger state space requires more iterations (2^n) to exhaustively test all input combinations. Additionally, the "reachable" method yields slightly faster simulation times, as it excludes a small number of unreachable states from analysis. However, the performance gain is marginal, since the majority of the state space remains reachable and must still be evaluated.

Controllers 1 and 2 scale nearly identically, with only minor differences in time due to measurement noise caused by other processes' interfering with the simulation on the computer (Fig 13). Controller 3 takes longer than the other two controllers for each bit size because its FSM has more states than those of Controllers 1 and 2, and as a result, the simulations have to run through more iterations (Fig. 14). The simulation time difference between Controller 3 and Controllers 1 and 2 becomes extremely noticeable for larger bit multipliers. For instance, for a 14-bit *mr*, the brute force simulation takes around 50 minutes for Controllers 1 and 2, whose FSMs both use a 5-bit state encoding ($2^5 = 32$ total states) (Fig. 13). Controller 3's brute force simulation for a 14-bit *mr*, by contrast, takes almost 110 minutes, as the FSM uses a 6-bit state encoding ($2^6 = 64$ total states) (Fig. 14). The brute force simulation time grows linearly with the size of the theoretical state space: as Controller 3's state space is two times larger than that of Controllers 1 and 2, the simulation time for an identically-sized *mr* input is also around twice as long.



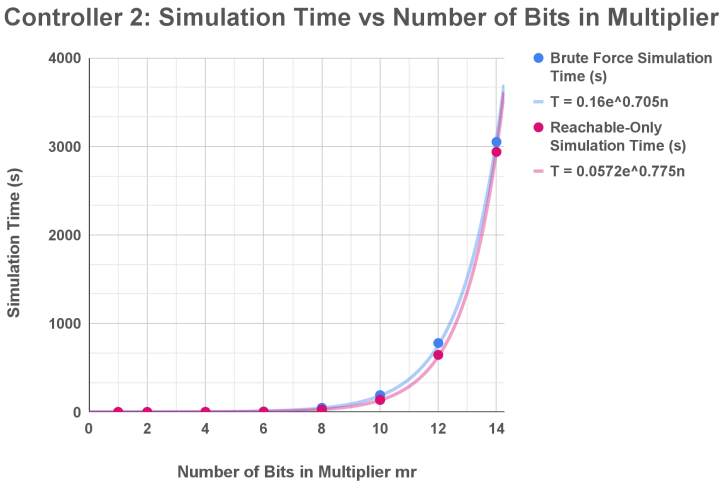


Figure 13: Timing analysis for Controllers 1 and 2: Results are almost identical as the number of states in FSM are the same

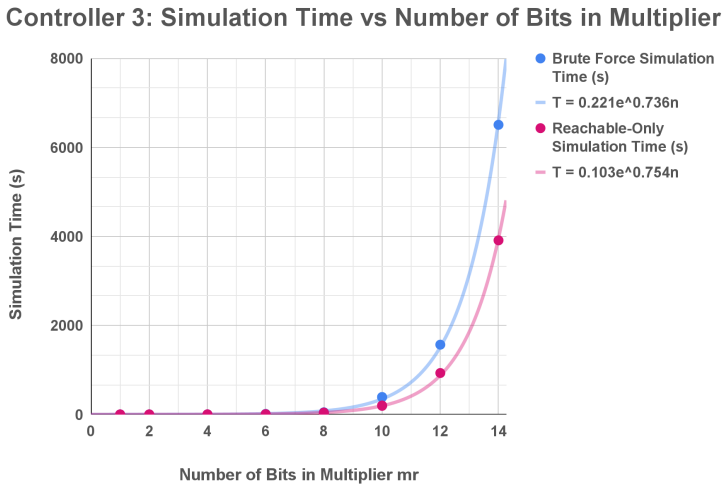


Figure 14: Timing analysis for Controller 3: Times required to complete simulations are higher because the FSM has more states

As the number of bits in the multiplier increases, the number of encoded states and input combinations grows rapidly, leading to a dramatic rise in the number of testbench iterations required to reconstruct the full FSM. This imposes a computational bottleneck, particularly for circuits with large input sizes or complex state behavior. While the "reachable-only" optimization

slightly mitigates this by pruning unused states, the marginal improvement highlights that most of the theoretical state space is still active in realistic controllers.

Modularity can help mitigate the exponential growth in simulation time by enabling verification of smaller components individually. By testing each submodule module for timing side-channel leakage in isolation, exponential timing growth may not be as serious of a problem, as most modules are quite small in scale. However, the exponential timing growth also emphasizes the need for more advanced optimizations, such as exploiting FSM symmetry or integrating early-exit conditions into the truth table extraction phase, to ensure the approach remains feasible for larger or industrial-scale designs.

5. Future Improvements and Optimizations

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

Currently, the algorithm employed first extracts the truth table and constructs the graphical representation of the FSM first. After, it runs the taint tracking with taint-kill algorithm to analyze whether there are secret-dependent differences in path lengths between pairs of dominator nodes. To improve efficiency, this process could be optimized by integrating the graph analysis directly into the truth table extraction phase, running it in real time within the Verilog testbench. If a difference in path lengths is detected early between any secret-dependent paths, the analysis can terminate immediately. This optimization may reduce the checking time in the best case.

Optimizations can also be made to avoid redundant analysis across identical or isomorphic subgraphs in an FSM, reducing computation time and improving scalability for larger input sizes. Perhaps repetitive patterns can be detected so that once a single iteration (for example, one bit's multiplication cycle) is analyzed and confirmed to be constant time, subsequent iterations with the same flow pattern can be declared constant time.

6. Conclusion

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.

Verilog was used as the HDL for this project, but the algorithm to extract the graphical FSM from the controller code is a general algorithm that could be implemented in any language to interface with any controller. Testbenches with the proposed algorithm can be written in another HDL and run simulations on controllers in that HDL to extract their FSMs regardless of their implementation details as well. By bypassing the limitations of syntax-dependent text parsing, this method proves to be robust across implementation styles and applicable to a wide range of controller designs. Furthermore, additional optimizations such as pruning unreachable states and leveraging structural patterns in FSMs allow for possible improvements in scalability and future developments in more automated and general formal verification methods for timing side-channel vulnerabilities in hardware systems.

References

[1] Kocher, P.C. (1996). Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: Koblitz, N. (eds) Advances in Cryptology — CRYPTO ’96. CRYPTO 1996. Lecture Notes in Computer Science, vol 1109. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-68697-5_9

[2] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom, “Spectre Attacks: Exploiting Speculative Execution,” in *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P ’19)*, 2019.

[3] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg, “Meltdown: Reading Kernel Memory from User Space,” in *Proceedings of the 27th USENIX Security Symposium (USENIX Security ’18)*, 2018.

[4] Sofia Marina, “Improving Taint Tracking Precision for Side Channel Vulnerability Detection,” Fall 2024.

[5] Chips Alliance, “Rocket Chip Generator,” GitHub repository, https://github.com/chipsalliance/rocket-chip, accessed Apr. 25, 2025.

[6] Free Chips Project, “Diagrammer: FIRRTL to GraphViz Visualizer,” GitHub repository. [Online]. Available: https://github.com/freechipsproject/diagrammer. Accessed: Apr. 28, 2025.

[7] Klaus von Gleissenthall, Rami Gökhan Kıçı, Deian Stefan, and Ranjit Jhala, “IODINE: Verifying Constant-Time Execution of Hardware,” in Proceedings of the 28th USENIX Security Symposium (USENIX Security ’19), Santa Clara, CA, Aug. 2019, pp. 1411–1428.

[8] Frank Vahid, *Digital Design with RTL Design, VHDL, and Verilog*, 2nd ed. Hoboken, NJ, USA: John Wiley & Sons, Mar. 9, 2010.

Appendix

1. Simulation timing data used to generate graphs in Figures 13 and 14:

Simulation Timing Data for Controllers 1, 2, and 3 with Varying Input Sizes

Number of Bits in mr	Controller 1 Brute Force Time (s)	Controller 1 Reachable Time (s)	Controller 2 Brute Force Time (s)	Controller 2 Reachable Time (s)	Controller 3 Brute Force Time (s)	Controller 3 Reachable Time (s)
1	0.0697967	0.0531564	0.0714115	0.0681194	0.0987116	0.0755834
1	0.0888619	0.0526742	0.0907973	0.0550615	0.0732149	0.06737
1	0.0873242	0.0618605	0.0868938	0.0631453	0.0842984	0.0653129
2	0.171837	0.109394	0.153394	0.145479	0.287972	0.149048
2	0.153631	0.142011	0.160495	0.135458	0.247651	0.146872
2	0.136015	0.138918	0.13202	0.115034	0.255919	0.152541
4	1.04413	0.709754	1.14161	0.645623	1.08885	0.889965
4	1.02735	0.718445	0.935787	0.620184	1.04314	0.937411
4	1.07149	0.593881	0.988644	0.642022	1.11226	0.921829
6	4.47635	4.07238	4.11233	3.7223	10.9144	6.13529
6	4.73575	4.1178	4.77589	3.67409	10.4829	5.77447
6	4.46447	4.44852	4.35983	3.98914	10.6805	6.14166
8	46.7344	27.3758	46.4679	26.6714	46.8307	39.2935
8	47.3966	26.8226	46.1239	26.7606	46.2154	40.0958
8	46.0088	27.5457	45.65	27.1338	47.6984	38.903
10	189.408	140.559	189.899	138.403	391.781	193.516
10	190.365	136.649	188.437	136.122	389.587	200.258
10	188.775	136.005	189.453	136.55	388.932	200.286
12	751.443	658.086	778.861	645.058	1567.9	931.258
14	3079.53	2974.29	3051.45	2938.42	6507.37	3911.94

2. Github repository containing all code: https://github.com/rf5447/ece398code