Experiment 9
Hardware Obfuscation

We describe an experiment to implement different Hardware Obfuscation techniques for securing hardware intellectual properties (IPs) against piracy and tampering attacks.

Case Study

The goal of this experiment is to learn the implementation and strength of two major types of hardware obfuscation.

In the semiconductor industry, the system-on-chip designers used to have fabrication facility inside their company. But as the feature size of transistor is shrinking, the establishment and maintenance cost of such facility sore so high that it is no longer feasible to have them inside each company. Rather, large fabrication facilities like TSMC fabricates chip from many fabless design houses.

In present day, with the adoption of horizontal business model, the chip integrator acquires third party intellectual property (IP) from around the world. This IP refers to a reusable unit of logic, cell, or chip layout design that is either licensed to another party or owned and used solely by a single party. Then the integrated design is sent to offshore DFT (design for test) insertion facility. The complete and verified design is sent to offshore fabrication facilities to production. The foundry also performs post silicon manufacturing test before slicing off the die. Then the tested chips are sent to another offshore facility for assembly. After assembly and testing, the final production reached the distribution network. The design houses have no authority over the facilities than the contract agreement.

Along this flow, the chip designs, both from the third-party vendor and from the system-on-chip designer, are vulnerable for piracy. IP can be sold in unauthorized manner, used in more instances than paid for, modified to contain malicious circuitry like hardware Trojans. The attacker, who can be a rogue employee in the facilities, can use reverse engineering to retrieve higher level design, and use that design in their chip claiming the design their own. For these reasons, the need for hiding the design or locking the functionality of chip is necessary. If the attackers are unable to extract the correct functionality of the chip, or cannot unlock the chip to function properly, they cannot perform the mentioned attacks.

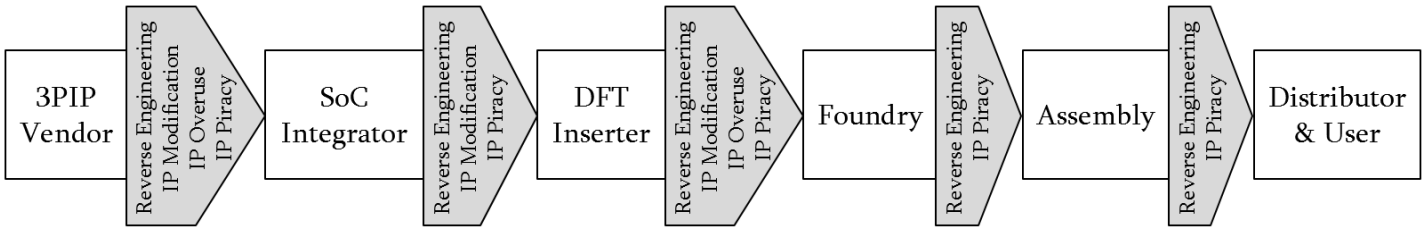

Figure 1: Threats in semiconductor supply chain

Theory Background

Hardware obfuscation is the method of hiding or locking the correct functionality of a chip, to protect the IP from various threats in supply chain.

1. Hardware Obfuscation

Hardware obfuscation is a method to hide the logic in a circuit design. By definition, obfuscation is the technique of obscuring or hiding the true meaning of a message or the functionality of a product, in order to protect the intellectual property inherent in the product.

The objective of hardware obfuscation is to have a functionally equivalent but structurally different design. The design is modified in a way that it implements different logic functions and it is not possible to retrieve the correct logic equation by reverse engineering. A locking mechanism must be incorporated which ensures the design becomes functionally equivalent upon correct unlocking process.

1.1 Combinational Obfuscation

The obfuscation that modifies only the combinational logic of a design is defined as combinational obfuscation. The most common method to do this type of obfuscation is inserting additional XOR/XNOR gates in the circuit. One input to these gates are connected to primary inputs, commonly termed as key inputs. From the truth table of XOR gate, we can see, when the key input is '0', Y=X and when key input is '1', Y = ~X. The opposite happens in XNOR gate. This can work as a lock, where the 'key' is the correct combination in key input.

Other techniques of combinational obfuscation include inserting 2X1 MUX gates or look-up-tables as locks, using reprogrammable logic to hide portion of logic, scrambling the data-paths etc.

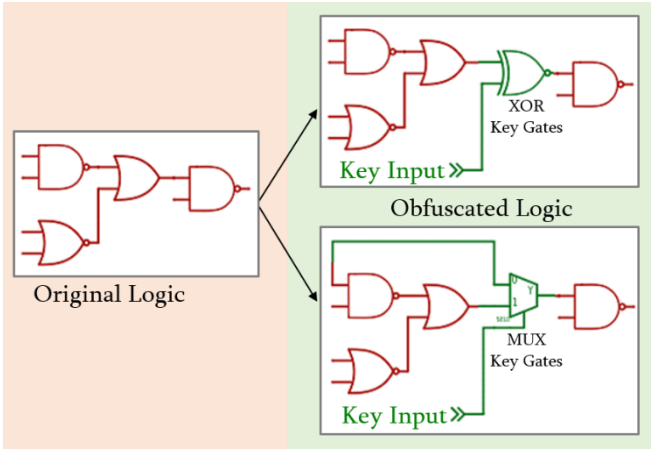

Figure 2: Combinational Obfuscation

Table 1: Truth table of XOR/XNOR Gate

		XOR Gate	XOR Gate
Key Input	X	Y	Y
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

1.2 Sequential Obfuscation

Practical ICs contain memory components like flip-flops, registers, latches. These memory elements constitute sequential circuit, in which logic at some certain point depends on the previous logic states. Memory elements along with combinational logic forms finite

state machines (FSM). In a state machine, the next state is determined by present state and inputs to the state machine.

The state machine of an IC can be obfuscated to lock the design. In regular state machine design, usually not all possible state is in operation. There are often states that are never occurs, or not reachable. These states can be used to introduce additional locking mechanism into the design. Another way to obfuscate state machine is hiding the startup state. One can add an extra state machine before the actual state machine which will make the access to the initial state of the original state reachable for only a specific input sequence.

For example, in Figure 3, The original FSM (blue) has startup state S_0^N. Without obfuscation, this the state the FSM starts with. The obfuscation FSM (green) is an additional FSM on the left which modified the startup state for the entire design to be S_0^O. Only if someone provide input in pattern P_1^O -> P_2^O -> P_3^O, he or she can reach the original startup state S_0^N. Once this state is reached, the FSM works as desired, or can be said 'unlocked'. Any other input sequence will render the FSM unusable as the original startup state cannot be reach.

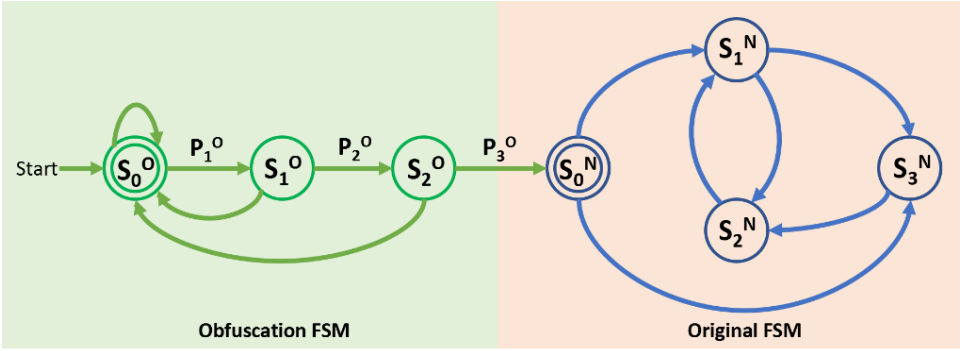

Figure 3: Sequential Obfuscation

An interesting modification of the obfuscation FSM is inserting blackhole state. A blackhole state is a state from which there is no exit path. Once a circuit reaches this state, it is stuck at that state forever until restarted. An example of this can be found in Figure 4. Blackhole states are effective for thwarting retries.

2 Brute Force Attack

There are a few attacks on hardware obfuscation. In this experiment, the brute force attack is observed. In brute force attack, an adversary tried to retrieve the correct key pattern by randomly applying key combinations. For this attack, the best-case scenario is when the attacker has to apply 2^N key patterns for a N bit lock.

Experiment Set-up: Configuration

The hardware and software needed for this experiment include:

1. The HAHA Board and Altera USB blaster.
2. A computer.
3. Altera Quartus platform to program the FPGA.

Knowledge Requirement:

1. Verilog HDL
    a. Netlist
    b. Testbench

For your convenience, example of simple verilog netlist and testbench is attached in the appendix section.

Instructions and Questions

In this experiment, you will perform obfuscation on circuits and observe how brute force attack can break obfuscation.

PART I Brute Force Attack on Obfuscation

1. Download the zip file marked as your group number. You will find a sof file which is designated only for your group.
2. Load the hex file to your HAHA board with Altera USB Blaster Device Programming.
3. Connect Slide Switch to the 5-bit key inputs so that MSB of key is assigned to Slide Switch [5] and LSB is assigned to Slide Switch [1]. Connect reset to Slide Switch [8].
4. Manually apply all key patterns. With each pattern, reset and see if the circuit is unlocked or not. For correct key, the 7-segment display will show a hex counter.
5. Demonstrate the working key to an instructor, or record a short video of the demonstration. Submit the detected key in report and the demonstration video. See the instruction for demonstration part for further information.
6. Take in image of the board showing the key pattern in switch box and any working digit on display. Submit the image in report.
7. Observe how long it took (approximately) to find a 5-bit key. Calculate how long will you take to unlock a 256-bit key (x 2^{256} / 2^{6}). Is it possible to break a 256-bit obfuscation manually with brute force attack? Submit your answer in report.

PART II Combinational Obfuscation

You are given a small benchmark file c499. You need to perform 16-bit combinational obfuscation on it and then verify the obfuscation with testbench.

1. All gates should be XOR/XNOR gates. The unlocking key should be the one assigned to your group in Table 2. The key input should be named as keyinput1, keyinput2, …. Submit the verilog file with report.
2. Write a testbench to test 10 random keys (each 16-bit) and the expected key. Choose 10 input patterns for each of the keys in the testbench (Use loops and $random). Submit the testbench with report.
3. Run simulation with testbench and take screenshots for one wrong key and the right key. Submit the screenshots in report.
   $For\ Input\ I, if\ Output\ O_w = f(I, Key_{wrong}), and\ O_r = f(I, Key_{right}), \quad then, O_w \neq O_r$
   The screenshot(s) should reflect this. You get to choose this arbitrary input.

PART III Sequential Obfuscation

You are given a small FSM design. The original state machine of this design is shown in Figure 4. You need to modify the FSM to have obfuscation FSM in the startup as presented in Figure 5. The input sequence for unlocking the FSM is unique for each group. Please follow Table 2 for your designated unlocking key.

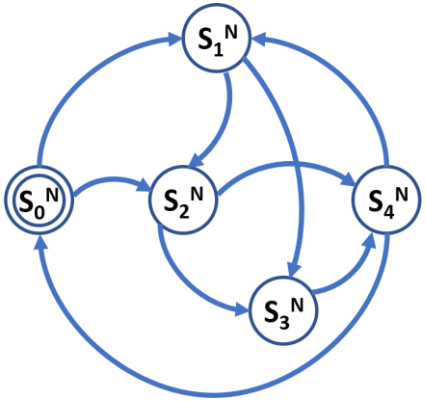

Figure 4: Original FSM

1. Design an obfuscation FSM with four states. Refer to Table 2 for the unlocking key P_1^O-P_2^O-P_3^O-P_4^O-P_5^O specific for your group. Submit the modified Verilog file with report.
2. Write a testbench with
   a. wrong key ~P_1^O->~P_2^O->~P_3^O->~P_4^O->~P_5^O
   b. wrong key P_1^O->~P_2^O->~P_3^O->~P_4^O->~P_5^O
   c. wrong key P_1^O->P_2^O->P_3^O->P_4^O->~P_5^O
   d. and right key P_1^O->P_2^O->P_3^O->P_4^O->P_5^O.

   Submit the testbench with report.
3. Run simulation. Display state registers so that blackhole state for wrong key can be viewed. At one run, you might want to comment out three other patterns and go with one unlocking state transition pattern. Because once you enter Blackhole state, there is no coming back to start-up state. So, for four patterns (three wrong, one right), you should have 4 screenshots. Submit the screenshots of simulation window in report.

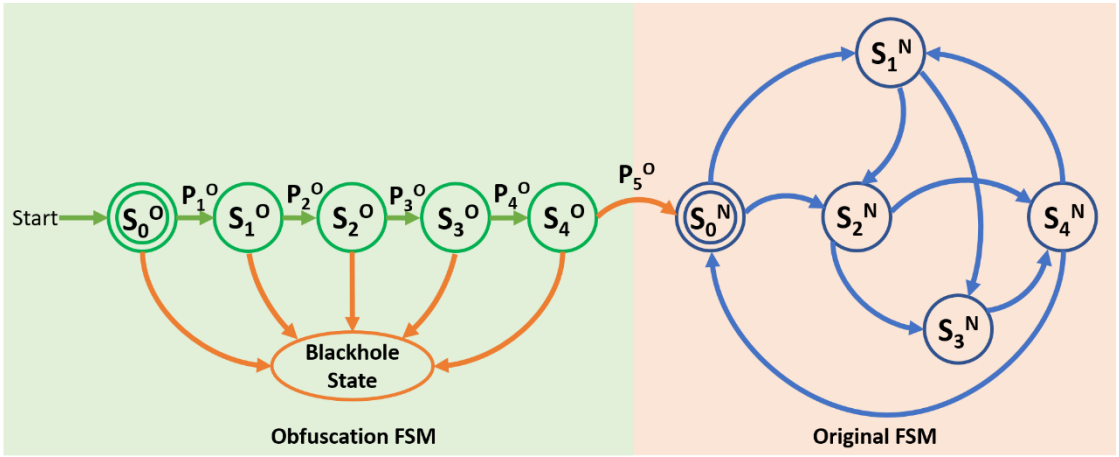

Figure 5: Original and obfuscation FSM for Part III

Table 2: Required unlocking keys for each group

Group No.	PART II Key	PART III Key
1	16'b0110110100100101	5'b10001
2	16'b0111100101011110	5'b11110
3	16'b1111101110101110	5'b01101
4	16'b0001101011010101	5'b11101
5	16'b1011111010100100	5'b01110
6	16'b1110101100100110	5'b01011
7	16'b0100001011010101	5'b10011
8	16'b1110000010100111	5'b01001
9	16'b0000011000111010	5'b00011
10	16'b1010011000010100	5'b11010
11	16'b0010001010010010	5'b00110
12	16'b0010100001101110	5'b11001
13	16'b1100100001000101	5'b10110
14	16'b1000100010010100	5'b00101
15	16'b1000100110101100	5'b10100

☞ Lab Report Guidelines

Deliverables:

1. For part I:
    a. Report the correct unlocking key (in report)
    b. Image of the board showing the key pattern in switch box and a working digit on display (in report)
    c. Calculation for question 7 (in report)
    d. Demonstration video (see instruction below)
2. For part II:
    a. Obfuscated verilog file
    b. Testbench
    c. Screenshot(s) showing simulation displaying I, O_w, Key_{wrong} and I, O_r, Key_{right} for same input pattern I.
3. For part III:
    a. Obfuscated verilog file
    b. Testbench containing three wrong key and the right key
    c. Screenshots of simulation.

Demonstration:

1. For part I:
   Once your figure out the correct unlocking key, make a short video showing
    a. A wrong key loaded in switch → reset → the 7-segment display is showing anything except a counter.
    b. The right key is loaded in switch →reset → the 7-segment display is showing a counter.

Optional Follow-up

Brute-force attack with testbench:

You will need to design a testbench that can apply all possible key patterns on an obfuscated circuit and compares the outputs with the original circuit to check if both matches implying the obfuscation is broken.

Design a testbench like figure 6 –

- with the combinational obfuscated c499 you generated in Part II, and
- the provided unlocked c499 circuit in Part II.

Figure 6 shows only one output. You will need to XOR each output of original circuit with corresponding output of obfuscated circuit. Whenever the output pair of an XOR matches, it would produce a 0. For correct key, all XOR output will be 0. You need to detect this condition. You can do that with a NOR gate with all these comparison XOR outputs. You can try a few inputs to compare the outputs to make sure obfuscation is broken.

In your testbech, using $fwrite, write the keys and the output of that NOR on a text file. Analyzing this file, you can see which one is the unlocking key for your obfuscation.

1. How long does it take to run the simulation?
2. Is there more than one key that could work as unlocking key?
3. Submit the testbench along with the report.

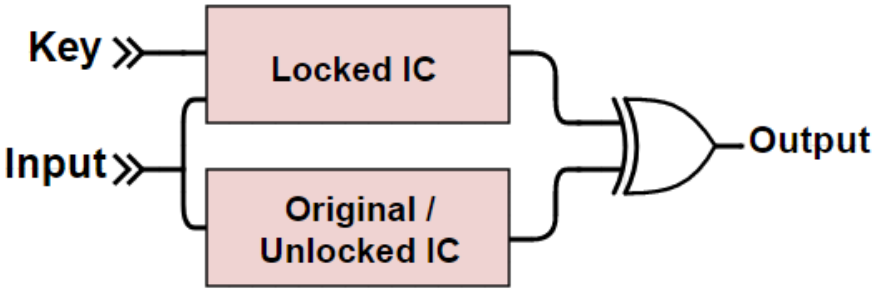

Figure 6: Automating Brute-force attack

References and Further Reading

[1] Forte, Domenic, Swarup Bhunia, and Mark M. Tehranipoor. "Hardware Protection through Obfuscation." (2017).

[2] Roy, Jarrod A., Farinaz Koushanfar, and Igor L. Markov. "Ending piracy of integrated circuits." *Computer* 43, no. 10 (2010): 30-38.

[3] Alkabani, Yousra, and Farinaz Koushanfar. "Active Hardware Metering for Intellectual Property Protection and Security." In *USENIX security*, pp. 291-306. 2007.

[4] Sarah Amir, Bicky Shakya, Domenic Forte, Mark Tehranipoor, and Swarup Bhunia. “Comparative Analysis of Hardware Obfuscation for IP Protection.” GLSVLSI

Appendix

A. Verilog Netlist Example:

```
module c17 (N1,N2,N3,N6,N7,N22,N23);

    input N1,N2,N3,N6,N7;

    output N22,N23;

    wire N10,N11,N16,N19;

    nand NAND2_1 (N10, N1, N3);
    nand NAND2_2 (N11, N3, N6);
    nand NAND2_3 (N16, N2, N11);
    nand NAND2_4 (N19, N11, N7);
    nand NAND2_5 (N22, N10, N16);
    nand NAND2_6 (N23, N16, N19);

endmodule
```

B. Verilog Testbench Example:

```
`timescale 1ns / 1ps

module tb_c432;
  integer f;
  reg [0:35] in ; // Inputs
  wire [0:6] out ; // Outputs

  c432 uut (
    .N1(in[0]), .N4(in[1]), .N8(in[2]), .N11(in[3]),
    .N14(in[4]), .N17(in[5]), .N21(in[6]), .N24(in[7]),
    .N27(in[8]), .N30(in[9]), .N34(in[10]), .N37(in[11]),
    .N40(in[12]), .N43(in[13]), .N47(in[14]), .N50(in[15]),
    .N53(in[16]), .N56(in[17]), .N60(in[18]), .N63(in[19]),
    .N66(in[20]), .N69(in[21]), .N73(in[22]), .N76(in[23]),
    .N79(in[24]), .N82(in[25]), .N86(in[26]), .N89(in[27]),
    .N92(in[28]), .N95(in[29]), .N99(in[30]), .N102(in[31]),
    .N105(in[32]), .N108(in[33]), .N112(in[34]), .N115(in[35]),
    .N223(out[0]), .N329(out[1]), .N370(out[2]), .N421(out[3]),
    .N430(out[4]), .N431(out[5]), .N432(out[6])); // Circuit

  initial
  begin
    f = $fopen("tb_c432_comp_random.txt", "w+b");
    $fwrite(f, "/* Result of testbench */ \n");
    #50
    repeat(10) begin
      in = $random;
      #50
      $fwrite(f, $time, " : input = %d, output = %d \n ", in, out);
    end
  end
  $fclose(f);
endmodule
```