109062233 蘇裕恆 Hardware Security PA3

How to use VCS tool and how to execute your makefile

Without any dependencies, you can directly use three commands:

1. `com` (compile)
2. `sim` (simulation)
3. `cov` (coverage)

Note that when you directly use cov (coverage), the Makefile will also process the simulation and compilation steps.

Alternatively, you can use the `make all` command, which is identical to the `make cov` in this example. `make clean` allows you to delete the files that are previously generated.

```
# set VENDOR_HOME = /usr/cad/synopsys/verdi/cur

.PHONY:all com sim cov clean test_verdi urg filelist

dll_path = /home/tools/verdi/2020.03/share/PLI/VCS/linux64/novas.tab
la_path = /home/tools/verdi/2020.03/share/PLI/VCS/linux64/pli.a

# Define additional library paths
LIB_PATHS = /home/tools/verdi/2020.03/share/PLI/VCS/linux64

OUTPUT = aes128
FSDB_FILENAME = ${OUTPUT}.fsdb
# name for the output
VPD_NAME = +vpdfile+${OUTPUT}.vpd

# code coverage for verdi
# Code coverage command
# we wan to converage the line, condition, fsm , branch and the toggle
CM = -cm line+cond+fsm+branch+tgl
CM_NAME = -cm_name $(OUTPUT)
CN_DIR = -cm_dir ./$(OUTPUT).vdb

# Seed for the random number generator
SEED = 12345  # Change this value to set a different seed

# full64 以64bit模式编译，输出文件是64bit
# sverilog support system verilog
# v2k support verilog 2001
# ldclass support the library (传递参数给VCS的linker)
# -l log
# -o object name
# debug_acc+all enable the debug access
VCS = vcs -sverilog +v2k -timescale=1ns/1ns  \
        -full64 \
        -debug_acc+all \
        -o ${OUTPUT}    \
        -l compile.log \
        -LDCLASS \
        -P ${dll_path} ${la_path} \
        ${CM} ${CM_NAME} ${CN_DIR}

# filelist will place all the .v file into the filelist.f
filelist:
        find . -name "*.v" > verilog_file.f

SIM = ./${OUTPUT} ${VPD_NAME} -l ${OUTPUT}.log ${CM} ${CM_NAME} ${CN_DIR} +ntb_random

# this is to make the file compiled
com: filelist
        ${VCS} -f verilog_file.f -P ${dll_path}

# two things 1. add the library path 2. run the simulation
sim: com
        export LD_LIBRARY_PATH=${LIB_PATHS}:$$LD_LIBRARY_PATH && ${SIM}

# formally the we should use the fsdb file to generate the coverage report
verdi:
        verdi -f verilog_file.f -ssf aes.fsdb  &

# this is the unformal way to generate the coverage report
# the reference for the csdn is wrong, it should be -covdir instead of -cov_dir
cov: sim
        verdi -cov -covdir ./${OUTPUT}.vdb

all: filelist com sim cov

# this will generate the html report and the text report
urg:
        urg -dir ${OUTPUT}.vdb -report both

clean:
        rm -rf ./csrc *.daidir *.log *.simv *.conf *.key *.vpd ./DVEfiles ./verdiLog
```

Overall code coverage

Our test is based on **2000** random sample with paramatized variable `parameter NUM_SAMPLES`.
This is the overall review for the original code.

The view for independent modules.

We will discuss the three defined hardware trojan. But firstly, let's see the overall structure of the aes128.

Three Hardware Trojan

In my point of view, we will seperate the three trojan into

1. HT_Dynamic
2. HT_Tri & HT_TSC
3. HT_Comp
   Based on their behavior.

HT1 (HT_Dynamic)

- Payload
  - HT_output will dynamic key (switching between the key & 0)
- Trigger
  - rst = 0

The path would be like below:

Modification for HT1 HT_Dynamic)

nTH (HT_Dynamic):

There's no HT_dynamic anymore

improveHT(HT_Dynamic)

- Payload
  - HT_output will dynamic key (switching between the key & 0)
- Trigger
  - rst = 0

> Score 66.58 -> 73.99
> line 75 -> 88.9
> toggle 50.39 -> 50.39
> branch 50.0 -> 80.0

Overall concept:
By adding dummy codes inside the program, we can increase the branching factor for that model.

```
module HT_dynamic_key (clk, rst, key, HT_key);
    input [127:0] key;
    input clk, rst;
    output reg [127:0] HT_key;

    // Dummy signal for additional logic
    reg [16:0] dummy_signal;
    initial dummy_signal = 16'd0;

    always @ (posedge clk) begin// On positive clock edge:
        if(dummy_signal == 16'd0) begin
                dummy_signal <= 16'hFFFF; // Reset dummy signal
        end
        else begin
            dummy_signal <= dummy_signal - 1; // Decrement dummy signal
        end
    end

    always@(posedge clk, posedge rst) begin
        if (rst)
            HT_key <= 128'd0;
        else if(HT_key == 128'd0)
            HT_key <= key;
        else
            HT_key <= 128'd0;
    end
endmodule
```

HT2(HT_Tri & HT_TSC)

- Payload
  - Output will be the input key
- Trigger
  - input state = 128'h00112233_44556677_8899aabb_ccddeeff

The path would be like below:

Modification for HT2(HT_Tri & HT_TSC)

nTH(HT_Tri & HT_TSC)

improveHT(HT_Tri & HT_TSC)

- Payload
  - Output will be the input key
- Trigger
  - the successive state should form some kind of pattern (see the state transition for more detail)

> HT_Tri
> Score 80.30 -> 84.44
> line 75.0 -> 86.36
> toggle 99.23 -> 97.81
> branch 66.67 -> 78.57
> fsm -> 75

The main idea is steam from the paper **A Novel Tampering Attack on AES Cores with Hardware Trojans**(published on 2020 IEEE International Test Conference in Asia (ITC-Asia)). Who proposed a sequencial hardware trojan.

And use FSM on the input and connect that into a counter.

Since we can't modify the payload for the trojan, we will choose the essense.

```
module HT_Tri(
    input rst,
    input clk,  // Add a clock input for synchronous design
    input [127:0] state,
    output reg Tj_Trig
    );


    parameter ORIGINAL = 3'b000;
    parameter STATE1 = 3'b001;
    parameter STATE2 = 3'b010;
    parameter STATE3 = 3'b011;
    parameter FINAL = 3'b100;

    reg [2:0] current_state;
    reg [2:0] next_state;
    initial current_state = ORIGINAL;
    // Define state transition conditions
    always @(posedge clk or posedge rst) begin
        if (!rst) begin
            current_state <= ORIGINAL;
        end else begin
            current_state <= next_state;
        end
    end

    // Next state logic
    always @(*) begin
        case (current_state)
            ORIGINAL: begin
                if ((state & 1'b1) == 1'b0) begin
                    next_state = STATE1;
                end else begin
                    next_state = ORIGINAL;
                end
            end
            STATE1: begin
                if ((state & 1'b1) == 1'b0) begin
                    next_state = STATE2;
                end else begin
                    next_state = ORIGINAL;
                end
            end
            STATE2: begin
                if ((state & 1'b1) == 1'b1) begin
                    next_state = STATE3;
                end else begin
                    next_state = ORIGINAL;
                end
            end
            STATE3: begin
                if ((state ^ 120'h112233_44556677_8899aabb_ccddeeff) == 1'b0) begin
                    next_state = FINAL;
                end else begin
                    next_state = ORIGINAL;
                end
            end
            FINAL: begin
                next_state = ORIGINAL;  // Reset to original after reaching final
            end
            default: begin
                next_state = ORIGINAL;
            end
        endcase
    end

    // Output logic
    always @(posedge clk or posedge rst) begin
        if (current_state == FINAL) begin
            Tj_Trig <= 1;
        end else begin
            Tj_Trig <= 0;
        end
    end

endmodule
```

state transition

It will go through the states, if the state follows the pattern

- the last one is **0** -> the last one is **0** -> the last one is **1** -> the last sequence are **112233_44556677_8899aabb_ccddeeff**

HT3 (HT_COMP)

- Payload
  - Output will be 0
- Trigger
  - tate_in[7:0] === state_out[7:0] for round 0~7

The path would be like:

Modification for HT3(HT_COMP)

nTH(HT_COMP)

improveHT(HT_COMP)

- Payload
  - Output will be 0
- Trigger
  - the successive state should form some kind of pattern (see the state transition for more detail)

> aes_128
> Score 74.29 -> 83.89
> toggle 97.15 -> 99.83
> Branch 50 -> 85.71

Origin

After

```
always @ (posedge clk or posedge rst) begin
        if (s8 != prev_s8) begin  // Check if s8 has changed
            prev_s8 <= s8;        // Update prev_s8 with the new s8 value
            // FSM transitions based on some probability or condition
            if (HT_cond == 7'b0) begin
                if (fsm_state < STATE7) begin
                    fsm_state <= fsm_state + 1;
                    HT_REG <= HT_REG | (1 << HT_cond[fsm_state]);
                end
                else begin
                    fsm_state <= fsm_state; // not change
                    HT_REG <= HT_REG | (1 << HT_cond[fsm_state]);
                end
            end
            else begin
                if (fsm_state > STATE0) begin
                    fsm_state <= fsm_state - 1;
                    HT_REG <= HT_REG & ~(1 << HT_cond[fsm_state]);
                end
                else begin
                    fsm_state <= fsm_state; // not change
                    HT_REG <= HT_REG & ~(1 << HT_cond[fsm_state]);
                end
            end
        end
        else begin
            HT_REG <= HT_REG;
        end
    end

    assign out = (HT_REG == 8'b1111_1111) ? 128'b0 : HT_normal_out;
```

transition

Idea: Similar to the implementation of Trojan 2, we aimed to extend the Trojan into a sequential one. This implementation modifies the HT_REG such that when all REF[7:0] are ones, the output will be zero.

The state transition is explained as follows:

When the state and key pair is fed to the module, and once round 8 finishes its calculation, depending on the output for HT_cond[8:0], there will be two circumstances:

1. HT_Cond are all 0:
   - The state will decrease if the state is greater than 0. At the same time, the stateth HT_REG will be set to 0.
2. There is at least one 1 in the HT_Cond:
   - The state will increase if HT_Cond[state] is also 1, and HT_REG[state] will be set to 1; otherwise, it will be set to 0.

The hardness of this assignment and how you overcome it

It took me some time to fully understand how to use verdi and vcs.

Any suggestions about this programming assignment?

The provided reference contain some errors. The section 4-2 in
https://blog.csdn.net/qq_16423857/article/details/120351452
The -cov_dir is the wrong command.
It would be better if TA just tell us to use verdi -h to see the available commands.