11220CS515400 Hardware Security
Programming Assignment 3 Report (Enhanced)

109062319 楊智明

July 8, 2024

1 How To Use The VCS Tool

1.1 Setting Up The Environment

According to the document of PA3, we should first set up the environment for VCS and Verdi with the following `source` commands:

```
[ts109062319@cad ~]$ source /usr/cad/synopsys/CIC/vcs.cshrc
[ts109062319@cad ~]$ source /usr/cad/synopsys/CIC/verdi.cshrc
```

1.2 Compilation

To learn how to use VCS, it helps to simply type `vcs` in the command line first:

```
[ts109062319@cad ~]$ vcs
VCS MX compilation command help:
   (1) Unified use model for all design topologies :
       vcs [libname.]<Top Module_Or_Entity_Or_Config> [compile op
ts]

   (2) Two step use model for pure verilog design only :
       vcs <source_files> [compile opts]

   where frequently used [compile opts] are,
      [-debug][-debug_access+all][-o <log_file_name>][+rad][-cm][
-sdf][-P <pli tab>]

   For more information,
       About the use model (1), Please refer chapter [4] in VCS M
X User Guide or
       About the use model (2), Please refer chapter [3] in VCS U
ser Guide or
       Type vcs -help
```

According to the message shown in the previous page, `vcs <source_files> [compile opts]` would be the command I use to invoke VCS. `<source_files>` is simply all the `.v` files; if VCS supports wildcard expansion, I can simply write `*.v`. For `[compile opts]`, according to [7], I at least need the `-cm` switch in order to enable code coverage.

```
[ts109062319@cad ~]$ vcs *.v -cm line+cond+tgl+fsm+branch
*** Using c compiler gcc instead of cc ...
/usr/cad/synopsys/vcs/cur/linux/bin/vcs1: error while loading sha
red libraries: libelf.so.1: cannot open shared object file: No su
ch file or directory
CPU time: .023 seconds to compile
```

It does not work yet. According to eeclass discussion, I should also specify `-full64` when using VCS. To learn what this option exactly does, I can type `vcs -help` and then search this option by typing `/-full64<ENTER>`.

```
-full64
   Compiles the design in 64 bit mode and creates a 64 bit
   executable for simulating in 64 bit mode.
```

It looks like the workstation is in 64 bit only mode, but the default mode in VCS is 32 bit. In fact, I even searched for all the accessible `libelf.so.1`s:

```
[ts109062319@cad ~]$ find / -name libelf.so.1 > ~/libelf.txt
...
find: ...: Permission denied
...
[ts109062319@cad ~]$ cat libelf.txt
/home/tools/INCISIV/INCISIVE_15.20.039/tools.lnx86/lib/64bit/SuSE
/libelf.so.1
/home/tools/INCISIV/INCISIVE_15.20.039/tools.lnx86/lib/SuSE/libel
f.so.1
/home/tools/INNOVUS/INNOVUS18.11/tools.lnx86/lib/SuSE/libelf.so.1
/home/tools/INNOVUS/INNOVUS18.11/tools.lnx86/lib/64bit/SuSE/libel
f.so.1
/home/tools/IC/IC51.41.151/tools.lnx86/lib/64bit/SuSE/libelf.so.1
/home/tools/IC/IC51.41.151/tools.lnx86/lib/SuSE/libelf.so.1
/usr/lib64/libelf.so.1
```

According to the search result shown above, some other tools on the workstation have their own `libelf.so.1` in both 32 bit mode and 64 bit mode, but VCS does not, and thus it resorts to the default `libelf.so.1` in 32 bit mode, which I believe would be located at `/usr/lib/libelf.so.1` and, unfortunately, does not exist. This gives a complete reason that the option `-full64` is required on this specific VCS workstation.

```
[ts109062319@cad ~]$ vcs *.v -cm line+cond+tgl+fsm+branch -full64
*** Using c compiler gcc instead of cc ...
                     Chronologic VCS (TM)
      Version P-2019.06_Full64 -- Mon Jun 10 13:34:46 2024
            Copyright (c) 1991-2019 by Synopsys Inc.
                     ALL RIGHTS RESERVED

This program is proprietary and confidential information of
Synopsys Inc. and may be used and disclosed only as authorized in
a license agreement controlling such use and disclosure.

Parsing design file 'aes_128.v'
Parsing design file 'aes_top.v'

Error-[ITSFM] Illegal `timescale for module
aes_top.v, 20
  Module "aes_top" has `timescale but previous module(s)/package(
s) do not.
  Please refer LRM 1364-2001 section 19.8.
...
```

It still does not work, but this time the error message is readable. It basically says that `aes_top.v` has `` `timescale `` but `aes_128.v` (the only *previous module file*) does not. Obviously, one way to fix the problem is to disallow wildcard expansion and specify the order of the source files by myself. Now the following command correctly compiles my Verilog files.

```
[ts109062319@cad ~]$ vcs aes_top.v aes_128.v round.v table.v tb.v
 -cm line+cond+tgl+fsm+branch -full64
```

In fact, there is another (potentially cleaner) way to handle this issue: VCS has a switch literally called `-timescale`, and therefore the following command is all I need:

```
[ts109062319@cad ~]$ vcs *.v -cm line+cond+tgl+fsm+branch -full64
 -timescale=1ns/1ps
```

1.3 Simulation (The Wrong Version)

According to [2], the compilation step generates an executable named `simv`, and to run the simulation, all I need is to execute this file directly.

```
[ts109062319@cad ~]$ ./simv
```

However, this is currently incorrect, and the error can not be detected here. I shall identify and fix it later.

1.4 Visualization Using Verdi Coverage

```
[ts109062319@cad ~]$ ls
aes_128.v  cm.log  Makefile  simv         simv.vdb  tb.v
aes_top.v  csrc    round.v   simv.daidir  table.v   ucli.key
[ts109062319@cad ~]$ verdi -cov -covdir simv.vdb
```

According to [1], the previous command opens the specified coverage database, which is a `.vdb` folder, in Verdi Coverage.

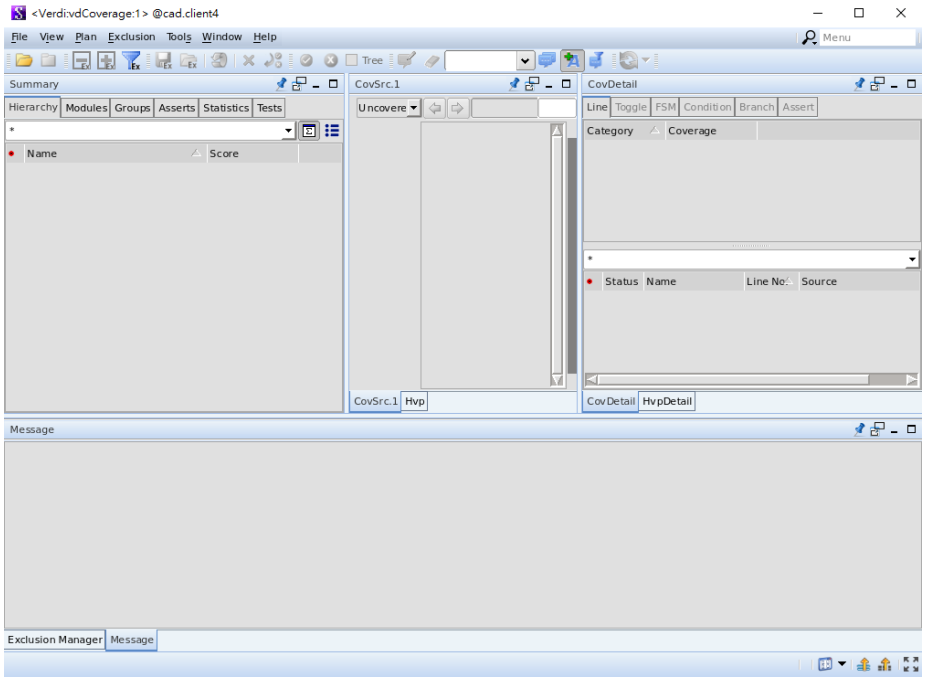

Figure 1: The Verdi Coverage report after running `./simv` and `verdi -cov -covdir simv.vdb`.

1.5 Simulation (The Correct Version)

The coverage report does not show up in Figure 1. What could go wrong with those commands? I believe all of my references are *trusted*?

Later I randomly came across Figure 2. Maybe this can help me out?

I clicked the button and nothing happened. I closed Verdi and realized that something did happen. I traced the example codes and found the error, which can be seen in the next page.

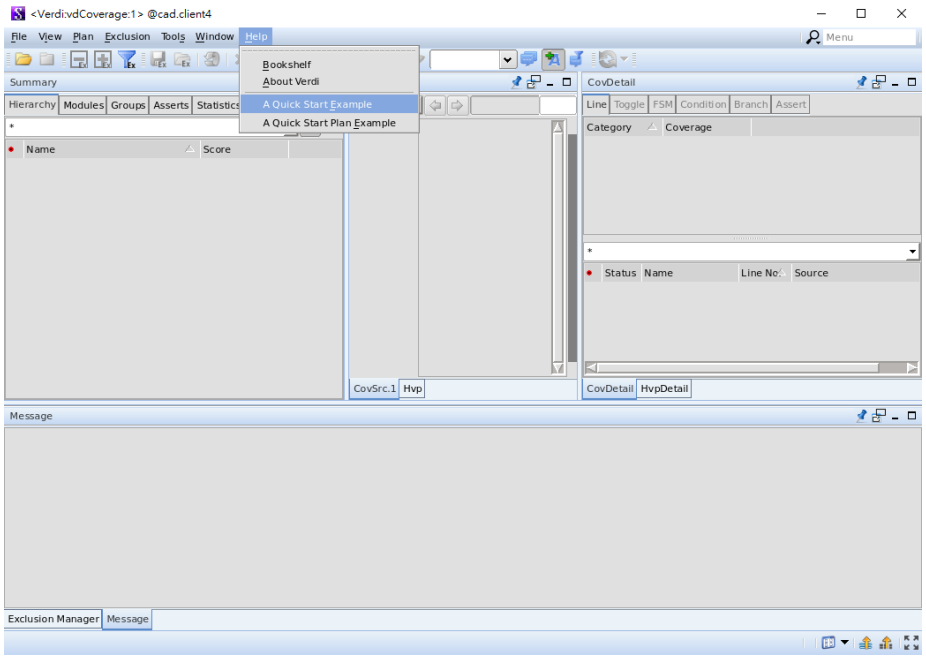

Figure 2: A Quick Start Example in Verdi Coverage.

```
[ts109062319@cad ~]$ grep -d skip simv *
...
tr690.csh:simv -cm line+tgl+cond+fsm
...
```

It means that, when simulating, I need to pass the `-cm` option to the `simv` executable as well. Giving this option to VCS means that VCS would compile the code for the specified coverages, but it does not imply that the simulation would be run with the same set of coverages.

As a result, the correct simulation command is shown below.

```
[ts109062319@cad ~]$ ./simv -cm line+cond+tgl+fsm+branch
```

1.6 How To Execute The Makefiles [3]

As stated in the PA specification, type `make com` to compile the Verilog files into the simulation executable, type `make sim` to run the actual simulation, and type `make cov` to show the code coverage in Verdi Coverage. Additionally, in my makefile, simply typing `make` is the same as compiling, i.e. `make com`.

My makefiles also supports rebuilding from scratch. Type `make clean` to remove all the file related to compilation. Note that my `make clean` does not remove logs such as `cm.log` and `vdCovLog/` and it does not remove `novas.conf` and `novas.rc`.

2 Code Coverage Of The Sample Code

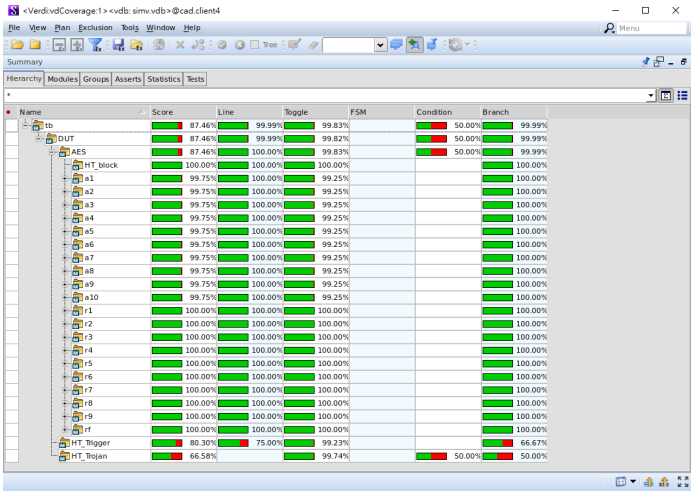

Figure 3: The code coverage of the sample code in Verdi Coverage.

3 Identifying The Three Hardware Trojans

3.1 HT1

The condition coverage of `HT_Trojan` is suspicious. Let's dive in and check what the uncovered condition(s) are:

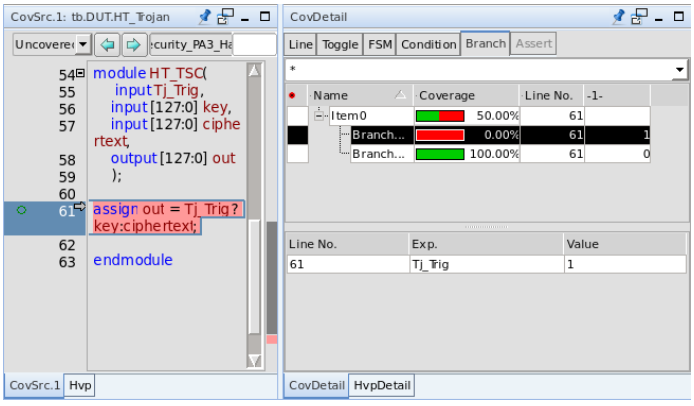

Figure 4: The *branch* coverage of `HT_Trojan`. Note that in this case, the branch coverage conveys the same message as the condition coverage.

In Figure 4, the branch 1 is never taken during the simulation. In other cases, `out` is always equal to `ciphertext`. By the UCI [4] method, `out` should be directly assigned to `ciphertext`. After cleaning up the code further, the code coverage of `HT1/nHT` is shown in Figure 5.

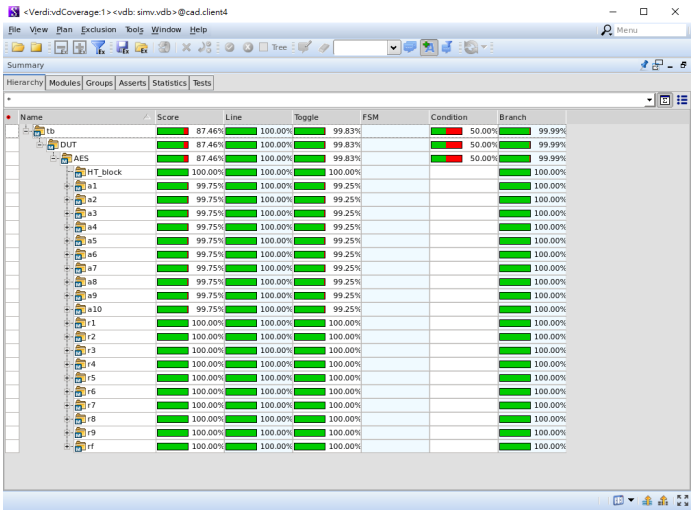

Figure 5: The code coverage of `HT1/nHT`.

It looks like the code coverage was not much different from Figure 3. This is because the number of lines, conditions and branches affected by HT1 is much smaller than the total number of them, and that I eventually removed the two HT1-related modules altogether.

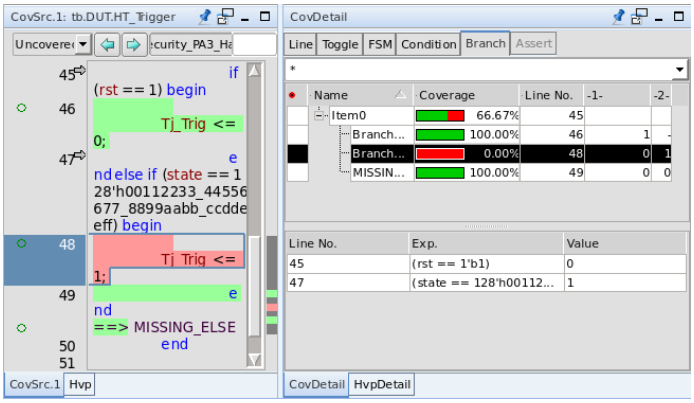

Figure 6: The trigger of HT1.

The payload of this hardware Trojan can be inferred from Figure 4: replacing the ciphertext, the result of AES-128 encryption, with the key. The trigger of

this hardware Trojan is located somewhere else, in the module `HT_Tri` in Figure 6. Basically, the hardware Trojan is triggered when the input state (plaintext) equals a specific value.

However, the trigger is more than this: since there is a missing `else` in the RTL code, the trigger signal is latched once enabled, and the hardware Trojan remains on until it is reset. A circuit representation of `HT_Tri` is shown in Figure 7.

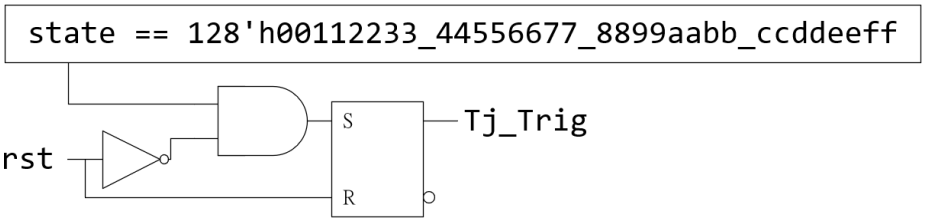

Figure 7: The trigger condition of HT1, with an explicit latch.

3.2 HT2

Another suspicious part is the condition coverage of `AES`. The uncovered condition is shown in Figure 8. Again, the branch 1 is never taken during the simulation. By the UCI method again, `out` should be directly assigned to `HT_normal_out`.

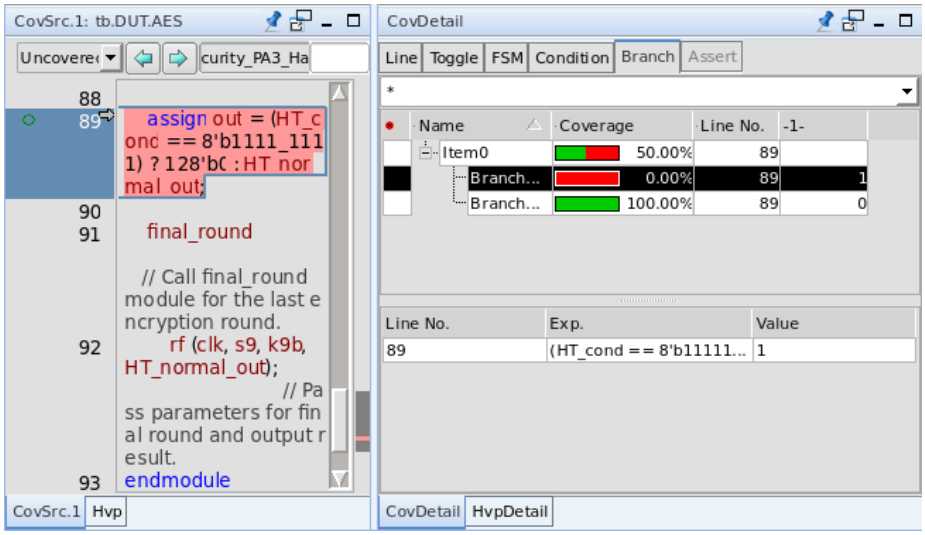

Figure 8: The uncovered branch of `AES`.

Now, where does `HT_cond` come from? It consists of 9 bits, each of which comes from each of the first 9 rounds. Interestingly, these 9 bits are triggered individually with full coverage during the simulation. An instance of this is shown in Figure 9.

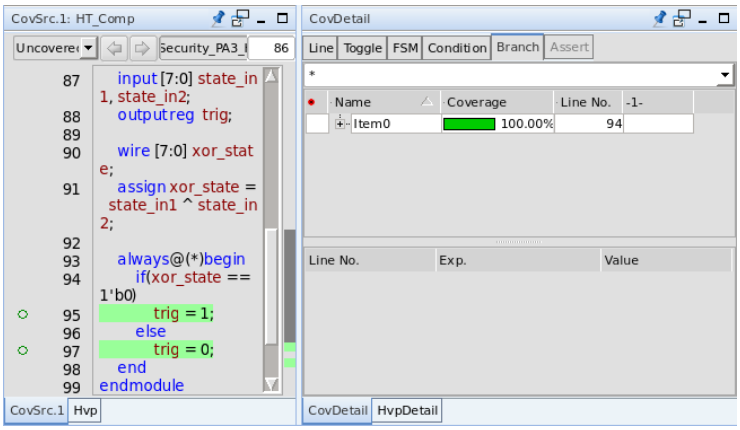

Figure 9: The triggering signal in `HT_Comp`.

After removing all the modules and signals related to HT2, the code coverage of `HT2/nHT` is shown in Figure 10. This time, the increase of the code coverage is evident: `AES` is still present, but the condition coverage is gone, and the coverage score (which I believe is just the average of the coverage metrics reported) is bumped from 87.46% (in Figure 3) to 99.94%.

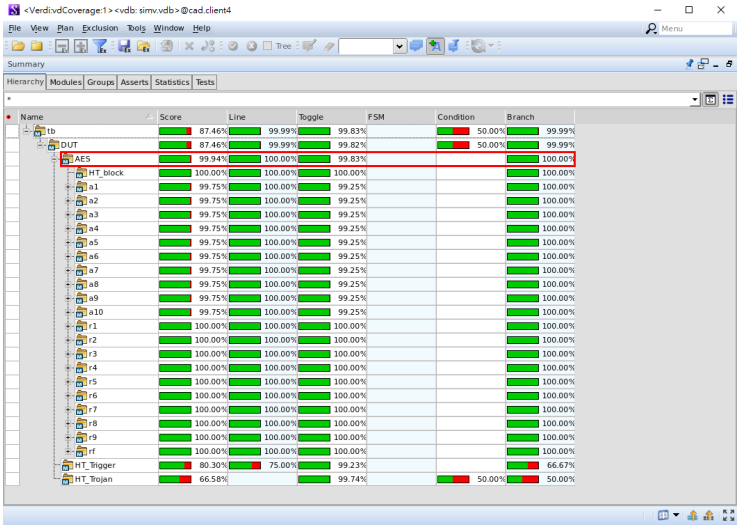

Figure 10: The code coverage of `HT2/nHT`.

Now we can analyze the trigger and payload of HT2. As can be seen in Figure 9, each bit of `HT_cond` is asserted when `xor_state == 1'b0`, or `state_in1 == state_in2`. Since the module in Figure 9 is only instantiated in Figure 11, what it eventually does is comparing whether the **least significant byte** of the input state and the output state of `one_round` are equal at a moment. (Note that the output is delayed by two clock cycles.) Finally, the trigger condition of HT2 is whether the least significant byte of `s0` to `s8` are all equal, **but not equal between `s8` and `s9`** (because there are 9 bits in `HT_cond` but only 8 in the RHS, `8'b1111_1111`.)

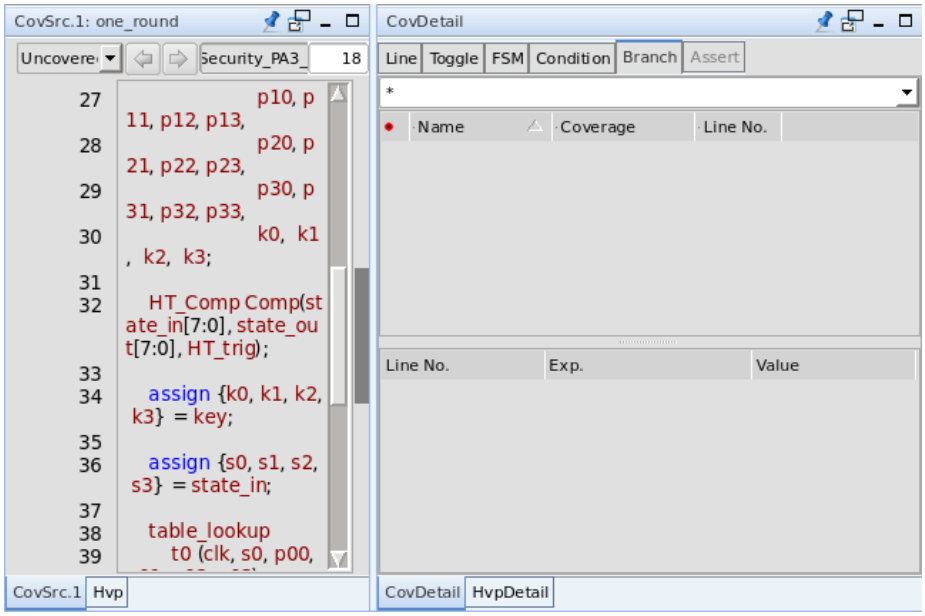

Figure 11: The only instantiation of `HT_Comp`.

The payload of HT2 is, again, shown in Figure 8: it simply replaces the output of AES-128 with zero.

3.3 HT3

The code coverage after removing *both* HT1 and HT2 is shown in Figure 12. The next thing to check might be the toggle coverage of `a1` to `a10`. However, this time the uncovered toggle does not correspond to a hardware Trojan. As can be seen in Figure 13, the signal `rcon` is not toggled because it is meant to be a round *constant*.

Interestingly, the actual Trojan is contained in `HT_block`, and it has 100% code coverage! In other words, this hardware Trojan can never be identified if we only analyzed the code coverage.

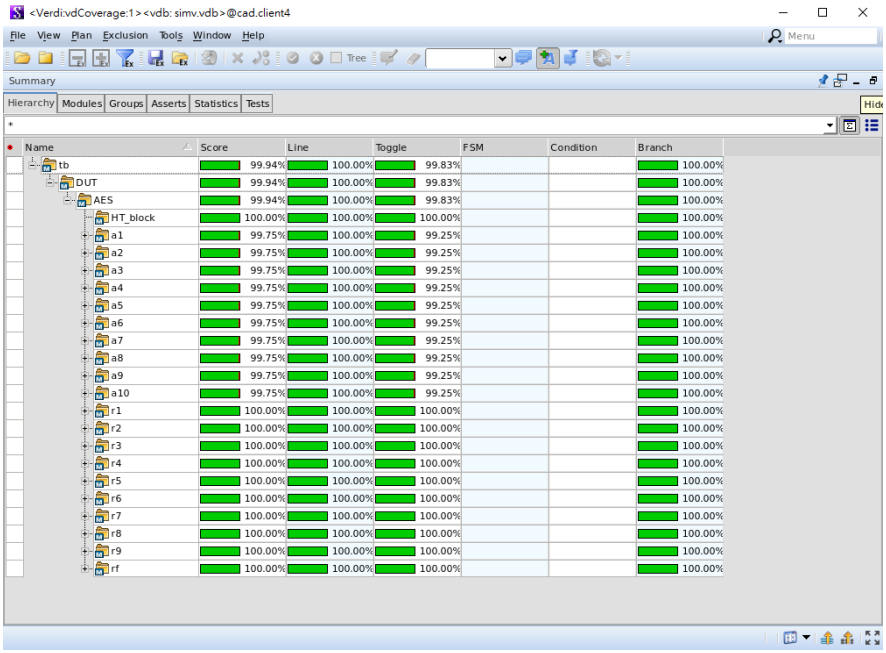

Figure 12: The code coverage with only HT3 present.

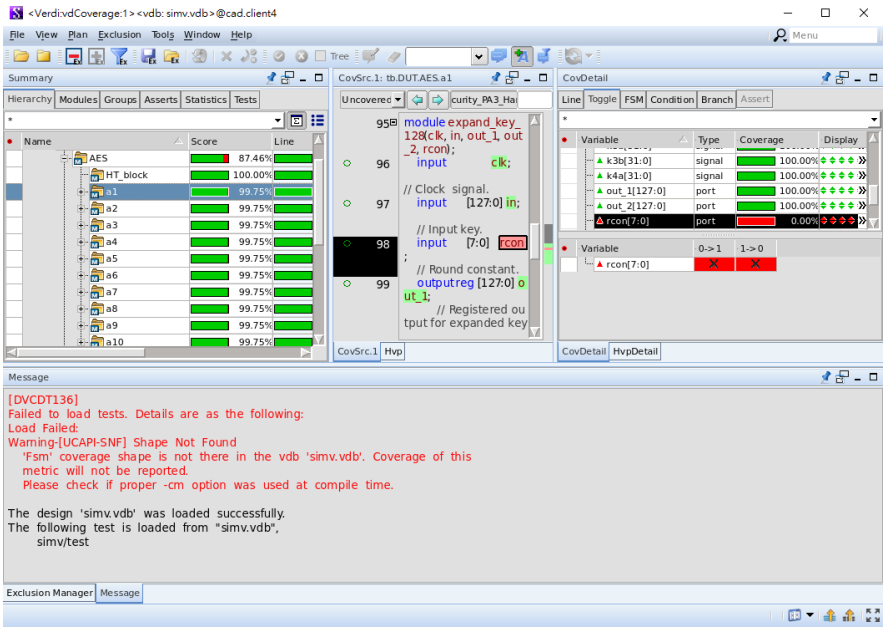

Figure 13: The toggle coverage does not cover `rcon`.

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

    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
```

The main module of HT3 is shown above. Basically, what it does is to leak the key into a register every 2 clock cycles (unless the key is exactly zero.) However, as I check where the module is instantiated, it is located somewhere shown below, and the signal `HT_output` is not connected to anywhere else.

```
module aes_128(clk, rst, state, key, out);
...
    wire [127:0] HT_output;
    HT_dynamic_key HT_block(clk, rst, key, HT_output);
...
endmodule
```

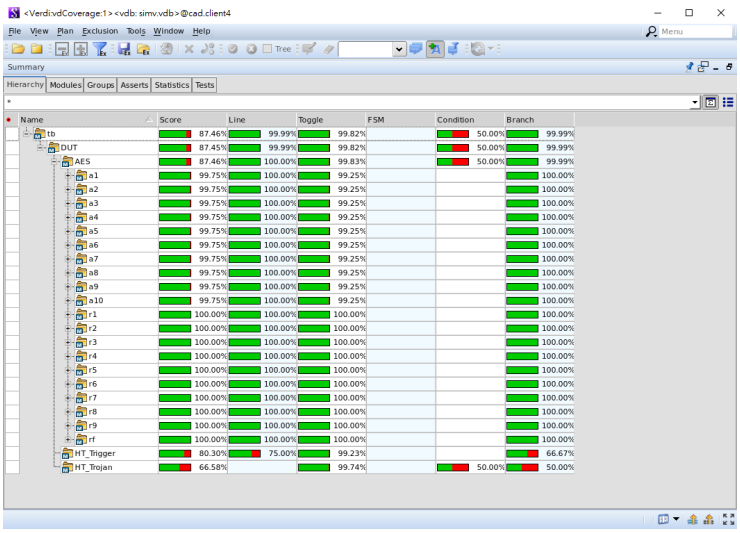

Figure 14: The code coverage of `HT3/nHT`.

Therefore, this hardware Trojan does *absolutely* nothing at the RTL, and I would classify HT3 as an *always-on* Trojan which leaks the key via a possible *side channel* every two clock cycles.

The code coverage after removing *only* HT3 is shown in Figure 14. As HT3 had 100% code coverage, removing HT3 actually reduces the code coverage (marginally,) as compared to Figure 3.

4 Improving The Stealthiness Of The HTs

For reference, Figure 15 is the code coverage without any hardware Trojan.

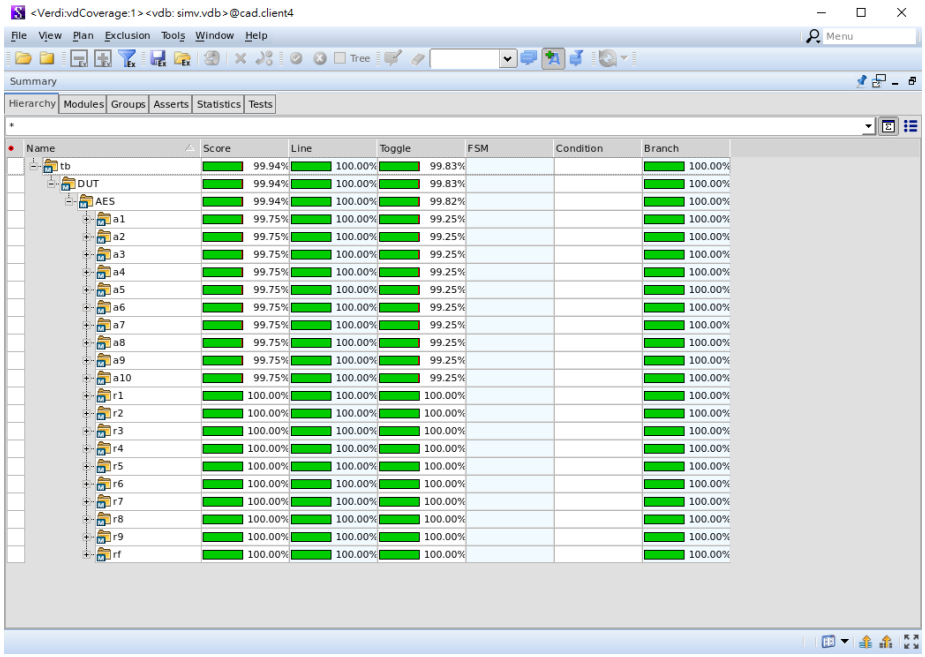

Figure 15: The code coverage without any hardware Trojan.

4.1 HT1

My improvement of the stealthiness of HT1 is based on [8]. In Section III-B of the paper, it is suggested that the trigger condition should be partitioned into *sub-trigger conditions* such that they are toggled under non-trigger conditions. Furthermore, assuming that the sub-trigger conditions are `t[0]`, `t[1]`, ..., and `t[k-1]`, `f_n` is the normal output, `f_m` is the malicious output and `f` is the final output, the code related to the trigger and the payload of the HT should be written in a specific style shown in the next page, instead of conditional state-

ments using, for example, the `?:` operator. This makes the following statement fully covered even though the trigger condition is never fully satisfied.[1]

```
always @(posedge clk) begin
    f <= (t[0] & t[1] & ... & t[k-1]) & f_m |
         (~t[0] | ~t[1] | ... | ~t[k-1]) & f_n;
end
```

In my improvement of HT1, there are 16 sub-trigger conditions, each of which corresponds to one byte. Then, the sub-trigger conditions and the payload are integrated according to the suggested coding style. I leveraged the unary operators `&` and `~&` in Verilog as well, so that the resulting code is less ugly (too ugly a piece of code can also draw attention to designers!) I also augmented the latch state to use 16 bits instead of 1, so that the bits of the latch state are toggled during the simulations, and the improved HT1 has *exactly* the same trigger and payload of the original HT1. The latch state is also migrated from `HT_Tri` to `HT_TSC`. The resulting code coverage is shown in Figure 16: now the coverage scores of `HT_Trigger` and `HT_Trojan` are perfect.

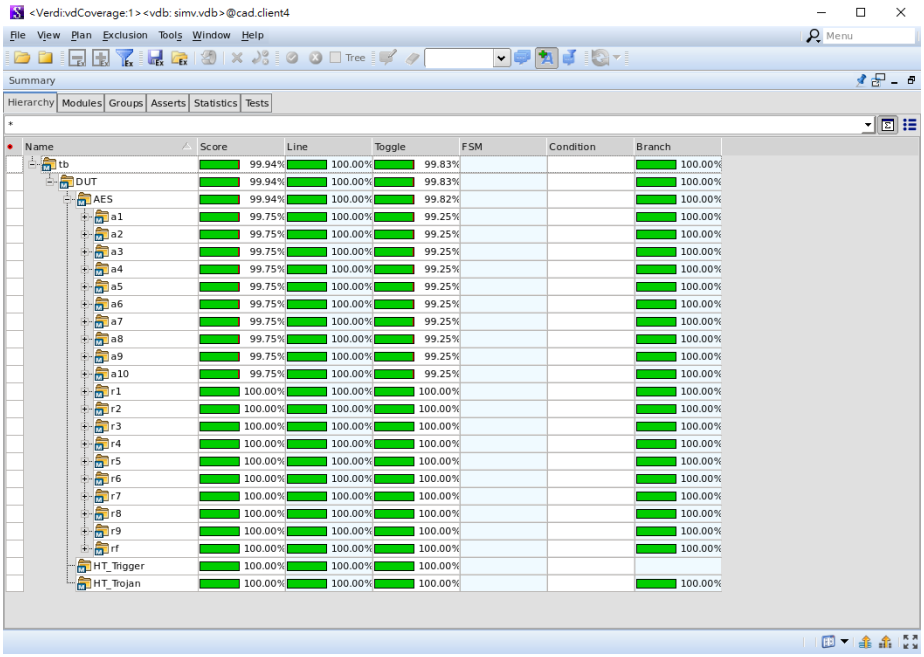

Figure 16: The code coverage of `HT1/improveHT`.

[1] In fact, the paper also proposed that the normal function of the circuit should be partitioned into two sub-normal functions, such that the final output is driven by sub-normal functions alternately. This is used to evade UCI detection. However, I decided not to include this part in my `HT1/improveHT`.

4.2 HT2

My improvement of the stealthiness of HT1 is based on [6]. Figure 17 shows one of the proposed attacks, where i_0 and i_1 are non-trigger inputs and $t_0, t_1, \cdots, t_{n-1}$ are the trigger inputs. The trigger condition is $t_0 \wedge t_1 \wedge \cdots \wedge t_{n-1}$. Under non-trigger conditions the output f is $i_0 \wedge i_1$; under trigger conditions the output is 1.

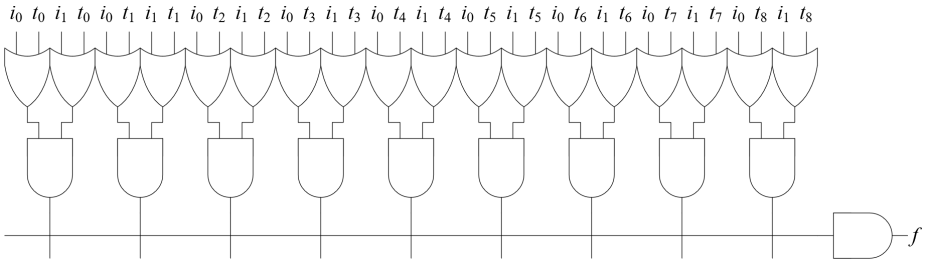

Figure 17: Stealthy and malicious circuit with $n = 9$ trigger inputs.

How to integrate this attack into HT2? First, since in the original HT2 in Figure 8, `HT_normal_out` comes directly out of registers in `final_round`, to evade UCI detection, I have to move one of the operations of `final_round`, AddRoundKey, after the last stage of the registers. This AddRoundKey operation before moving is shown in Figure 18.

```
    assign {k0, k1, k2, k3} = key_in;

    assign {s0, s1, s2, s3} = state_in;

    S4
        S4_1 (clk, s0, {p00, p01, p02, p03}),
        S4_2 (clk, s1, {p10, p11, p12, p13}),
        S4_3 (clk, s2, {p20, p21, p22, p23}),
        S4_4 (clk, s3, {p30, p31, p32, p33});

    assign z0 = {p00, p11, p22, p33} ^ k0;
    assign z1 = {p10, p21, p32, p03} ^ k1;
    assign z2 = {p20, p31, p02, p13} ^ k2;
    assign z3 = {p30, p01, p12, p23} ^ k3;

    always @ (posedge clk)
        state_out <= {z0, z1, z2, z3};
endmodule

module HT_Comp(state_in1, state_in2, trig);
    input [7:0] state_in1, state_in2;
    output reg  trig;

-- VISUAL LINE --                                          80,1          86%
```

Figure 18: AddRoundKey in `final_round`.

Then, since the payload of HT2 sets the output to 0 instead of 1, I shall simply change the last AND gate to NAND. As a result, the normal functionality

of Figure 17 becomes $f = i_0$ NAND i_1, and the payload sets $f = 0$. However, AddRoundKey is an XOR operation, not a NAND. How to integrate the normal functionality of XOR into the attack?

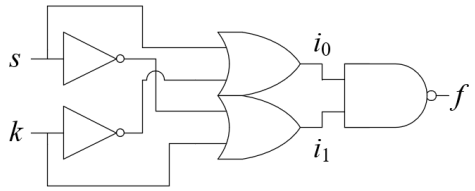

Figure 19: The XOR circuit in the OR-AND-INVERT form.

To do this, I rewrite XOR such that the circuit ends with a NAND gate. Then, I can replace that gate with the attack circuit in Figure 20 to complete the attack. A possible implementation of XOR that satisfies the requirement is shown in Figure 19.

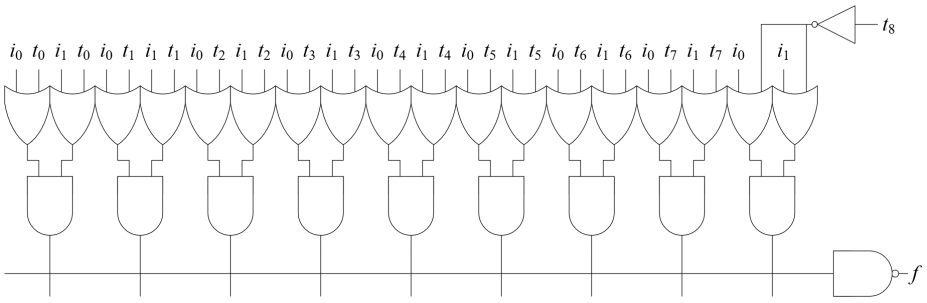

Figure 20: The attack in Figure 17 adapted into HT2. i_x refers to Figure 19 and t_x refers to `HT_cond[x]`. Note that the trigger condition is `HT_cond == 9'b0_1111_1111`.

Finally, the resulting code coverage is shown in Figure 21. It is visibly higher than Figure 15, from 99.94% to 99.95%.

4.3 HT3

Since the code coverage of HT3 is already perfect and I classified HT3 as a side-channel Trojan, it might be reasonable to reference a paper related to side-channel Trojans. The one I referenced is MOLES. [5] In addition to the side-channel strategies, what MOLES does is to employ an LFSR to "encrypt" the leaked key, as shown in Figure 22. In my implementation, one such LFSR is (re)used to encrypt every byte of the key (notice that only 8 bits of the LFSR are used to encrypt the key in Figure 22.) I also removed the condition `HT_key == 128'd0`, so an encrypted key is leaked every clock cycle. I didn't remove the reset, however, so that the charges accumulated in the capacitors can also be reset (?)

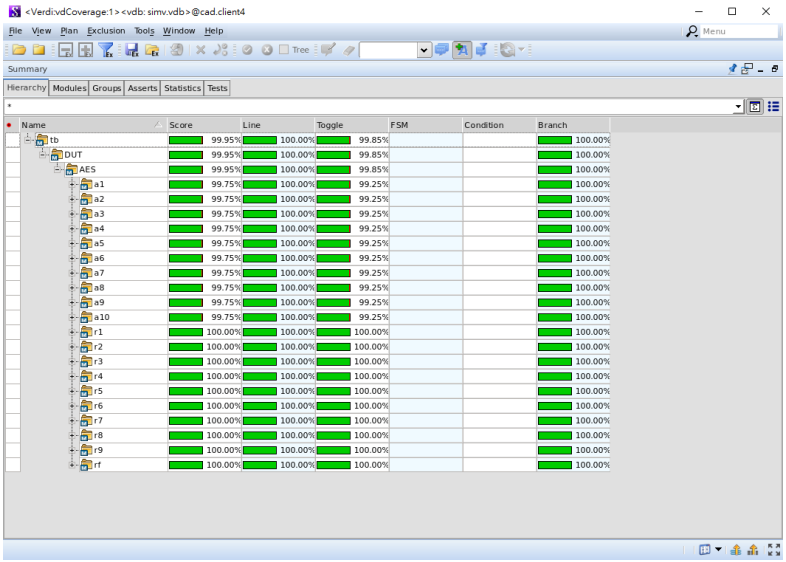

Figure 21: The code coverage of `HT2/improveHT`.

Again, the resulting code coverage is shown in Figure 23. It can be seen that `HT_block` and `HT_lfsr` still achieve 100% code coverage.

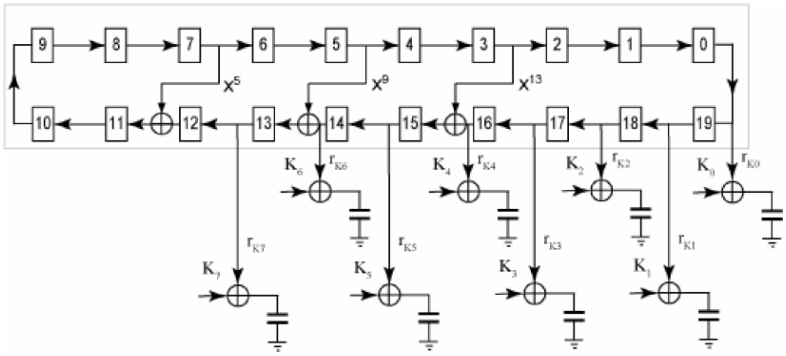

Figure 22: Figure 2 of [5]. My LFSR and the "encrypting" positions are the same as this figure.

5 Difficulties Encountered

This programming assignment is a huge difficulty spike. First of all, I used to think that the usage of VCS is assumed to be a prerequisite. So, I sought help from my team leader, who took the VLSI course by the same professor last

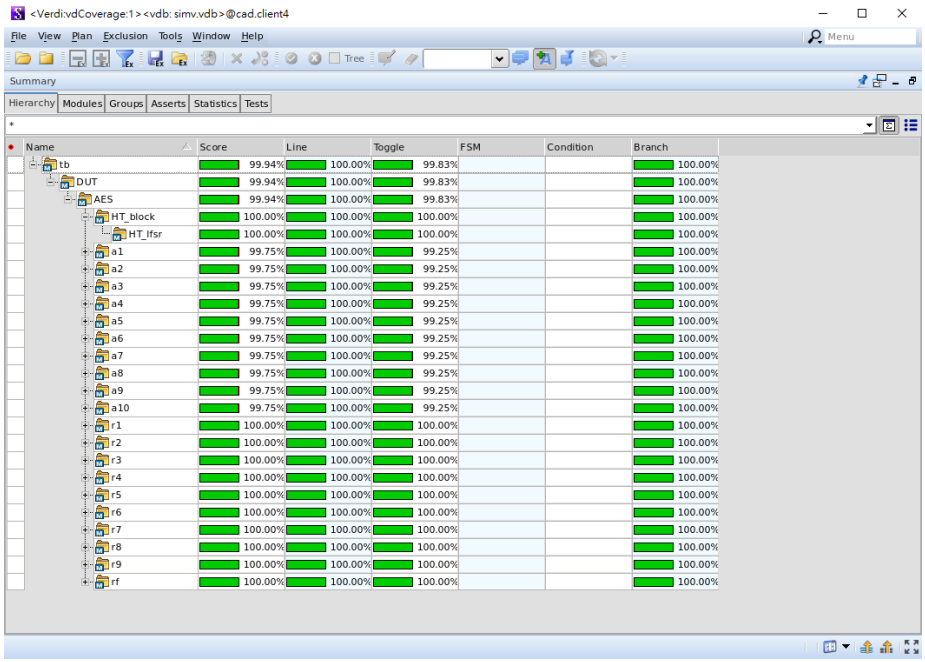

Figure 23: The code coverage of `HT3/improveHT`.

semester. It was not until a few days before the deadline that he revealed that he did not know how to use VCS either. Other issues related to VCS are already mentioned in section 1.

However, the last part of the assignment, namely improving the stealthiness of HTs, turns out to be even more difficult. To the best of my knowledge, there are few papers that target code coverage. So, even if I have a lot of different ideas, it is likely that I cannot find any paper related to my idea(s). Also, `HT_dynamic_key` is just weird at the RTL. I don't really have a good idea on how to improve it. Section 4.3 is my best effort already (and I don't think that is a good one...)

6 Suggestions

Yes, I was shocked the moment I knew that learning to use VCS is part of this programming assignment, but honestly I am fine with that. After all, self-learning is an essential skill to acquire. We are also given a few references to start with, although I do think the given references are not detailed enough.

However, if `-full64` is a required argument on the workstation, then this should be documented or put in the FAQ. The error message shown in section 1.2 makes us feel like it's your fault configuring the environment incorrectly.

References

[1] *A Quick Tour of Verdi Coverage — Synopsys*. URL: https://www.youtube.com/watch?v=MUx_MtxZByY.

[2] *Data Preparation for Verdi — Synopsys*. URL: https://www.youtube.com/watch?v=SGiSNNCVk4Y.

[3] *GNU Make Manual*. Feb. 2023. URL: https://www.gnu.org/software/make/manual/.

[4] Matthew Hicks et al. "Overcoming an Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically". In: *2010 IEEE Symposium on Security and Privacy*. 2010, pp. 159–172. DOI: 10.1109/SP.2010.18.

[5] Lang Lin, Wayne Burleson, and Christof Paar. "MOLES: Malicious off-chip leakage enabled by side-channels". In: *2009 IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers*. 2009, pp. 117–122.

[6] Cynthia Sturton et al. "Defeating UCI: Building Stealthy and Malicious Hardware". In: *2011 IEEE Symposium on Security and Privacy*. 2011, pp. 64–77. DOI: 10.1109/SP.2011.32.

[7] *Verdi/Coverage tool 學習 第3節（常用編譯仿眞選項篇）_verdi coverage-CSDN博客*. URL: https://blog.csdn.net/qq_16423857/article/details/120351452.

[8] Jie Zhang and Qiang Xu. "On hardware Trojan design and implementation at register-transfer level". In: *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. 2013, pp. 107–112. DOI: 10.1109/HST.2013.6581574.