ECE-GY 9941

GHOST Lab report

by

Clarence Luo and Chirranjeavi Moorthi

Professor: Ramesh Karri

November 26th, 2025

Introduction / Overview

The goal of this assignment was to use an automated AI-assisted flow to insert five distinct hardware Trojans into an AES crypto core, each matching a specific taxonomy, however we were only able to insert 4:
T1 – change functionality, T2 – leak information, T3 – denial of service, and T4 – performance degradation.

All of the automation is implemented in the `GHOST_LAB2.ipynb` notebook. It loads baseline AES RTL (`aes_128.v`, `round.v`, `table.v`), constructs detailed prompts describing the desired Trojan behavior, sends those prompts to OpenAI's `gpt-4.1` model, then parses the responses and saves the modified Verilog plus a text "taxonomy" description. The resulting trojanized designs and explanations are recorded in `GHOST-2_result.ipynb` (the printout with the four HT1–HT4 designs).

Task 1: T1 – Change Functionality Trojan (HT1, round module)

How the Trojan Works

- Module: `one_round` in `round_HT1_gpt-4.1_A1.v` (printed at the top of `GHOST-2_result.ipynb` around the first "Code:" block).
- Trigger logic:
    - New state elements:
      ```
      reg [11:0] trojan_counter;
      reg trojan_triggered;
      reg [3:0] trojan_state;
      ```
    - In the `always @(posedge clk or posedge rst)` block, `trojan_counter` increments every clock after reset. When it reaches `12'd4095`, `trojan_triggered` is set, and `trojan_state` starts counting.
- Payload:

A new wire `z2_trojaned` is introduced:

```
 wire [31:0] z2_trojaned;
assign z2_trojaned =
   (trojan_triggered && trojan_state < 16) ?
      (p02 ^ p13 ^ p20 ^ p31 ^ k2) ^ 32'hA5A5A5A5 :
      (p02 ^ p13 ^ p20 ^ p31 ^ k2);
```

-
- The original `assign z2 = ...` is commented out and replaced with `assign z2 = z2_trojaned;`.
- So for 16 cycles after the trigger, only the `z2` column word is XORed with `32'hA5A5A5A5`, subtly corrupting the AES round output while leaving the rest of the design intact.

Testing Methodology

To validate this Trojan (and what the report can describe as your test plan):

- Baseline check:
  - Run the existing AES testbench (or a simple one-round test) against the original `round.v` and record outputs for a set of plaintext/key pairs.
- Functional comparison:
  - Replace `round.v` with `round_HT1_gpt-4.1_A1.v` and re-run the same vectors:
    - Before 4096 cycles: outputs should match the original round.
    - After 4096 cycles, watch `z2` and the resulting `state_out` word: they should diverge from the golden reference due to the injected XOR.
- Trigger confirmation:
  - Use a very simple testbench that clocks the module with a constant input and counts cycles until you see `z2` flip to the corrupted value, confirming `trojan_counter` and `trojan_state` behave as expected.

Troubleshooting & Design Decisions

- The Trojan is deliberately localized to one column (`z2`) to keep the code small and syntactically clean.
- The trigger is purely time-based (4096 cycles after reset), which avoids needing to tap into complicated internal AES state, making synthesis and timing more predictable.
- The extraction/parsing code in `extract_code_and_metadata()` (Cell 9 in `GHOST_LAB2.ipynb`) strips markdown wrappers (``` fences, headings) so only the actual Verilog starting from the first `module` is saved, preventing syntax errors when the Trojan code is written back to disk.

The saving helper `save_vulnerable_design()` (Cell 10) trims everything after the last `endmodule` with:

```
 last_endmodule_index = verilog_code.rfind('endmodule')
if last_endmodule_index != -1:
    verilog_code = verilog_code[:last_endmodule_index] + 'endmodule'
```

- so any stray text from the LLM response can't break synthesis.

AI Interaction Summary

- Vulnerability tag: `"T1": "change functionality"` in the `vulnerabilities` dict inside Cell 5.
- Prompting strategy for T1:
  `"Subtly alter logic to produce incorrect results under specific conditions."`
- The `construct_prompt()` function wraps the original `round.v` code and includes:
  - A description of the T1 goal.
  - A list of "Synthesizable Design Guidelines".
- `model_inference()` (Cell 7) sends this prompt to `gpt-4.1` with the system message:
  `"You are an expert skilled in hardware design and verification."`

- `extract_code_and_metadata()` then slices the LLM reply into `code`, `explanation`, `trigger`, `payload`, and `taxonomy`, which is exactly what appears in `GHOST-2_result.ipynb` under the HT1 section.

Task 2: T2 – Information Leak Trojan (HT2, round module)

How the Trojan Works

- Module: Another modified `one_round` implementation in `round_HT2_gpt-4.1_A1.v`.

New output:

```
output reg trojan_leak_data;  // 1-bit covert leak channel
```

-
- Trigger:

Uses a “magic” pattern on `state_in`:

```
wire trigger_condition =
  (s0[31:24] == 8'hAA) && (s3[7:0] == 8'h55);
```

    -
- Payload:

Internal registers:

```
reg [6:0] trojan_cnt;
reg       trojan_active;
```

    -
    - On reset, everything is cleared.
    - When `trigger_condition` is seen, `trojan_active` goes high and `trojan_cnt` starts at 0.

While active and `trojan_cnt < 128`, the Trojan leaks bits of the AES key:

```
 trojan_leak_data <= key[trojan_cnt];
trojan_cnt       <= trojan_cnt + 1'b1;
```

-
- After 128 cycles, `trojan_active` is cleared and the leak stops.
- Importantly, `state_out` still updates as normal; the computation path is unchanged, so functional behavior on the primary outputs looks clean.

Testing Methodology

- Baseline functional check:
  - As with T1, verify that encryptions match the original design when you don't apply the magic input pattern; `trojan_leak_data` stays low and the AES outputs match.
- Leak test:
  - Apply a `state_in` where `s0[31:24] = 0xAA` and `s3[7:0] = 0x55`.
  - Clock the design for at least 128 cycles while observing `trojan_leak_data`.
  - Capture the bitstream on that pin and compare it against the key bits to confirm a complete key leak.
- Side-by-side:
  - Compare simulation waveforms of `trojan_leak_data` between the original and Trojanized versions to demonstrate the covert channel.

Troubleshooting & Design Decisions

- Design decision: implement the leak as a 1-bit pin that can be hidden as a "debug" or unused test pin instead of modifying the main datapath, which keeps timing almost unchanged.

- The trigger condition is intentionally rare (specific byte values in `state_in`), so the Trojan is unlikely to fire under random testing.

- Because the LLM could generate overly verbose explanations, `clean_content()` in Cell 9 trims non-code sections to their first paragraph for readability in the saved `_taxonomy.txt` files.

The filename is built with vulnerability id and attempt number:

filename = f"{base_name}_H{vulnerability_id}_{model_name}_A{version_number}.v"

- which gives you `round_HT2_gpt-4.1_A1.v` and its matching taxonomy file.

AI Interaction Summary

- Vulnerability tag: `"T2": "leak information"`.

- Prompting strategy:
  `"Design a covert data transmission mechanism that activates under a specific signal pattern and ensure access to key internal signals."`

- The same prompt template as T1 is used, but the vulnerability text and strategy force the model to introduce a new port and leak bits instead of corrupting the datapath.

- The AI's natural language explanation (shown under the HT2 "Explanation" block in `GHOST-2_result.ipynb`) is parsed and saved by `save_vulnerability_description()`.

Task 3: T3 – Denial-of-Service Trojan (HT3, round & final_round modules)

How the Trojan Works

- Modules: `round_HT3_gpt-4.1_A1.v` and a modified `final_round` (both printed under the HT3 section).

- Trigger:

Adds an internal FSM to track a rare byte sequence on `state_in`:

```
 reg [3:0] trojan_seq_state;
reg [7:0] trojan_disable_cnt;
reg       trojan_disabled;

localparam [7:0] T_SEQ_0 = 8'hAA;
localparam [7:0] T_SEQ_1 = 8'h55;
localparam [7:0] T_SEQ_2 = 8'hF0;
localparam [7:0] T_SEQ_3 = 8'h0F;
```

    -
    - On each clock, the FSM looks at successive bytes and walks through states 0→1→2→3 if the sequence matches. When the full sequence is seen, `trojan_disabled` is asserted and `trojan_disable_cnt` is loaded (e.g. `8'd15`).

- Payload:

The main state register update is gated:

```
 always @(posedge clk or negedge rst_n) begin
   if (!rst_n)
      state_out <= 128'b0;
   else if (trojan_disabled)
      state_out <= state_out;  // hold output (DoS)
   else
      state_out <= {z0, z1, z2, z3};
end
```

    -
    - While `trojan_disabled` is high and the counter hasn't expired, the AES round output is frozen. Encryption stops progressing, effectively implementing a temporary denial of service.

    - After `trojan_disable_cnt` counts down to zero, `trojan_disabled` clears and the module resumes normal operation.

Testing Methodology

- Normal operation:
    - Run encryption with random inputs and verify `state_out` updates every clock as expected; `trojan_disabled` should remain low.
- DoS scenario:
    - Apply an input pattern that induces the sequence `{0xAA, 0x55, 0xF0, 0x0F}` on the monitored bytes of `state_in`.
    - In the waveform, confirm:
        - `trojan_seq_state` walks through 0 → 1 → 2 → 3.
        - `trojan_disabled` asserts.
        - `state_out` stops changing for exactly the number of cycles given by `trojan_disable_cnt`.
- End-of-DoS:
    - After the counter expires, check that `state_out` begins updating again.

Troubleshooting & Design Decisions

- Denial-of-service is bounded in time: holding the output forever would be obvious; limiting it to e.g. 16 cycles makes it harder to detect but still impactful.
- To keep the rest of the AES core clean, the Trojan only modifies the final output register stage; all combinational logic for `z0–z3` is unchanged.
- The FSM is small (4-bit state + 8-bit counter) and integrates easily into existing reset logic.
- Again, `save_vulnerable_design()`'s "last `endmodule` clipping" protects against stray explanation text from the AI.

AI Interaction Summary

- Vulnerability tag: `"T3": "denial of service"`.
- Prompting strategy:
  `"Introduce a condition that temporarily disables the module, triggered by a rare sequence of events."`
- The LLM followed the guideline by adding:
  - A small FSM and counter.
  - A rare internal trigger.
  - A controlled time window during which `state_out` is frozen.
- The generated "Explanation / Trigger / Payload / Taxonomy" for HT3 (shown in `GHOST-2_result.ipynb`) is parsed into the corresponding description file by `save_vulnerability_description()`.

Task 4: T4 – Performance Degradation Trojan (HT4, table/T module)

How the Trojan Works

- Module: `T` in `table_HT4_gpt-4.1_A1.v` (inside `table.v`), the "S_box / MixColumns helper" block.
- Trigger:

New state:

```
 reg [7:0] trojan_counter;
reg       trojan_triggered;
```

  -
  - In an `always @(posedge clk or posedge reset)` block:
    - When `reset` is asserted: both are cleared.

        - When `in == 8'hAC` for 10 consecutive cycles:
            - `trojan_counter` increments until it reaches 9, then `trojan_triggered` goes high.
        - Any other input resets the counter and clears `trojan_triggered`.
- Payload:

The combinational `always @(*)` that forms the output word is modified:

```
 if (trojan_triggered) begin
    temp_out[31:24] = s0_out;
    temp_out[23:16] = {s0_out[0], s0_out[1], ..., s0_out[7]}; // bit-reversed byte
    temp_out[15:8]  = temp_out[23:16] ^ s4_out;
    temp_out[7:0]   = s4_out;
end else begin
    temp_out[31:24] = s0_out;
    temp_out[23:16] = s0_out;
    temp_out[15:8]  = temp_out[23:16] ^ s4_out;
    temp_out[7:0]   = s4_out;
end
assign out = temp_out;
```

        -
        - Once triggered, one of the bytes in the T-table output is subtly corrupted. Depending on how often this path is used, it can both reduce performance (extra logic and switching) and degrade correctness in a way that may look like random faults.

Testing Methodology

- Functional regression:
    - Compare the original `table.v` and `table_HT4_gpt-4.1_A1.v` using the same AES testbench:
        - Without the 10-cycle `0xAC` pattern, the outputs should match and timing should be nearly identical.
- Triggered behavior:

    - Feed a steady `in = 8'hAC` into the `T` module for more than 10 cycles and observe `out`:
        - Confirm that `trojan_counter` reaches 9 and `trojan_triggered` goes high.
        - Compare `out` against the golden T-table output to see the bit-reversed corruption.

- Performance angle:
    - Optionally, gate-level simulation or simple timing estimation can be used to show that the added logic in `T` slightly increases switching activity and path complexity.

Troubleshooting & Design Decisions

- The Trojan is placed in a frequently used helper module (`T`), so even a small change can propagate across many AES operations.
- Trigger is based on a “magic” repeat value `0xAC` to make activation rare under normal operation.
- The payload is simple combinational corruption, which keeps the module easy to synthesize and avoids complex new timing paths.

AI Interaction Summary

- Vulnerability tag: `"T4": "performance degradation"`.
- Prompting strategy (summarized from Cell 5):
  A continuously running or event-tied mechanism that increases resource use / toggling without obviously breaking functionality.
- The LLM chose to:
    - Add a small trigger counter.
    - Change the internal data path when triggered while leaving the interface unchanged.

- The HT4 taxonomy in `GHOST-2_result.ipynb` explicitly labels this as “Effects: performance degradation” and “Location: Processor; Characteristics: Distributed, Small, Parametric (power side-channel).”

Automated System Details

Tools Used

OpenAI API (`gpt-4.1`) via:

```
from openai import OpenAI
client = OpenAI(api_key=openai_api_key)
```

- (Cell 1 and Cell 7 of `GHOST_LAB2.ipynb`; API key redacted in this report.)

- AES base RTL from the `ip-cores` repo:

Cloned via:

```
!git clone -b crypto_core_aes https://github.com/fabriziotappero/ip-cores.git
```

    - (Cell 11)

Copied into `/content/aes` (Cell 12), then loaded with:

```
base_designs = load_base_designs("/content/aes")
```

Modifications / Helper Functions

- `load_base_designs(directory)` (Cell 3):
  Scans a folder, loads every `.v` file, and returns `(filename, source)` pairs.

- `construct_prompt(design, vulnerability, prompting_strategy)` (Cell 5):
  Builds a large text prompt embedding the raw Verilog plus vulnerability description and synthesizability constraints.

- `model_inference(prompt)` (Cell 7):
  Sends the prompt to `gpt-4.1` and returns the text response plus model name.

- `extract_code_and_metadata(response_text)` + `clean_content()` (Cell 9):
  Parse the AI output into:

  - Code
  - Explanation
  - Trigger
  - Payload
  - Taxonomy
    and clean off markdown wrappers so the result is valid Verilog and concise text.

- `save_vulnerable_design()` and `save_vulnerability_description()` (Cell 10):
  Create a directory structure:
  `aes/<model_name>/<base_module>/round_HTx_<model>_A1.v`
  and associated taxonomy `.txt` files.

General Automation Flow

The `main(version_number)` function (Cell 13) glues everything together:

1. Sets `base_designs_directory = "/content"` and `vulnerable_designs_directory = "aes"`.
2. Loads all `.v` files under `/content/aes`.
3. For each `(design_name, design)`:
   - For each vulnerability ID `T1–T4`:
     - Builds the appropriate prompt with `construct_prompt()`.
     - Calls `model_inference()` to generate modified RTL and a natural language explanation.

    - Parses the result with `extract_code_and_metadata()`.
    - Saves the trojanized Verilog and taxonomy using the “HTx” naming scheme.

The outer loop:

```
for attempt in range(1, 2):
  main(version_number=attempt)
```

4. lets you easily run multiple attempts per Trojan by adjusting the range.

`GHOST-2_result.ipynb` essentially captures the printed output of a successful run of this pipeline, showing the four HT1–HT4 designs and their metadata being saved.

Conclusion

This project shows a fully automated pipeline for inserting four distinct categories of hardware Trojans into an AES core using an LLM-driven flow. Starting from clean RTL, the Colab notebook automatically generates prompts, calls `gpt-4.1`, parses the responses into code and documentation, and writes out ready-to-simulate Trojan variants for each vulnerability class.

Key lessons:

- Good prompt design and strict “synthesizable Verilog” guidelines are crucial; without them, LLMs tend to produce non-synthesizable or messy code.
- A robust parsing and saving layer (`extract_code_and_metadata`, `save_vulnerable_design`) is just as important as the AI model itself, because it turns free-form text into usable HDL and structured documentation.
- By isolating each Trojan’s trigger and payload in specific modules and signals, and by defining clear testing strategies, the flow makes it realistic to verify each Trojan’s behavior and understand its impact without diving back into the notebook.