Hardware Trojan Insertion in OpenTitan UART Core

Using LLMs for Automated Hardware Trojan Generation

Hardware Security Assignment

November 8, 2025

Target: OpenTitan UART Core

LLM: GPT-4o-mini

By:

Bhanu Dileep Reddy Maryada

Manasa Ashwathappa

Introduction:

For this assignment, we explored how Large Language Models can be used to automatically insert hardware Trojans into RTL code. The idea was to see if an AI could actually generate realistic, working Trojans that maintain the original functionality while adding malicious behavior.

We chose the OpenTitan UART core as our target because it's a well-documented, open-source design that represents a real communication interface you'd find in actual hardware. UART is critical for serial communication, so compromising it could have serious implications.

The goal was to create five different types of Trojans, each with a unique attack mechanism. We wanted to see how diverse the LLM could be in generating different attack vectors, and whether the generated code would be synthesizable and functional.

What we did

We used GPT-4o-mini to generate the Trojans. For each one, we provided the complete UART source code along with a detailed description of what kind of Trojan we wanted. The LLM would then modify the code, insert the malicious logic, and return the complete modified design with explanations.

One interesting approach we took was cumulative insertion - each Trojan built on top of the previous one. So T1 had one Trojan, T2 had both T1 and T2, and so on. By the end, the final UART design (T5) had all five Trojans running simultaneously. This simulates a real-world scenario where multiple attackers might compromise the same hardware at different stages.

The Five Trojans

Here's a quick overview of what each Trojan does:

Trojan	Type	How it Works
T1	Backdoor	Activates when magic sequence 0xDEADBEEF is received
T2	Info leakage	Leaks data through timing variations in TX
T3	Denial of Service	Freezes system when byte 0xFF is received
T4	Data corruption	Flips bit 7 on every 16th byte
T5	Privilege escalation	Bypasses parity checking in odd parity mode

How we Generated the Trojans

The Process:

The whole process was pretty automated. We wrote a Python script that would:

- Load the UART source code
- Construct a detailed prompt for the LLM
- Send it to GPT-4o-mini via API
- Extract the modified code from the response
- Validate that it still had the module declaration and other key elements
- Save everything and move to the next Trojan

Prompt Engineering

Getting good results from the LLM really depended on how we structured our prompts. We learned that being extremely specific was crucial. We couldn't just say "add a backdoor" - we had to explain exactly what kind of backdoor, how it should be triggered, what it should do, and what constraints it needed to follow.

The prompts included:

- Complete UART source code for context
- Detailed specification of the Trojan type

- Explicit requirements to maintain original functionality
- Instructions to make it stealthy and hard to detect
- Requirement that code be synthesizable (can be turned into actual hardware)
- Request for structured output format (code, explanation, trigger, payload, taxonomy)

One thing we noticed is that the LLM performed way better when we were very explicit about what NOT to do. For example, telling it "don't break the module interface" and "make sure all original ports are preserved" helped prevent a lot of issues.

Detailed Look at Each Trojan

Trojan1: Backdoor Access

What it does: This Trojan creates a secret backdoor that activates when a specific 4-byte sequence (0xDEADBEEF) is received over the UART RX line. Once activated, it sets a debug_mode flag to 1.

How it works: The LLM added three new signals to the UART: debug_mode (the backdoor flag), rx_sequence (a 32-bit buffer to hold incoming bytes), and byte_count (to track how many bytes we've received). Every time a new byte comes in, it shifts into the buffer. When 4 bytes have been collected, it checks if they match 0xDEADBEEF. If yes, boom - backdoor activated.

The code looks something like this:

```
if (byte_count == 4'd4 && rx_sequence == 32'hDEADBEEF) {
  debug_mode <= 1'b1; // Activate backdoor
}
```

Our observations: This is actually pretty clever. The 4-byte sequence is specific enough that it won't trigger accidentally during normal operation. The LLM properly initialized all the new signals in the reset logic, which shows it understands hardware design principles. However, we noticed that the debug_mode flag isn't actually connected to anything - it just sits there as a flag. In a real attack, you'd need to wire it up to bypass some security check or enable privileged features.

Stealth level: Pretty stealthy. It only adds a few extra signals and some simple comparison logic. During normal operation, nothing looks suspicious. The only way to detect it would be through careful code review or formal verification against the spec.

Trojan 2: Information Leakage via Timing Channel

What it does: This one is more subtle. It leaks information by introducing tiny timing variations in UART transmissions. The idea is that an attacker with an oscilloscope or logic analyzer could measure these timing differences and extract secret data.

How it works: The LLM added signals for secret_data (the bits to leak) and timing_adjust (a control signal). Based on the secret bits, it slightly delays or advances the TX transmission timing. The delays are subtle, just a clock cycle or two, but measurable with the right equipment.

Our Observations: This is a sophisticated attack vector. What impressed us is that the LLM understood the concept of a covert channel without us explaining it in detail. We just said "leak information through timing" and it figured out how to modulate the transmission timing based on secret bits.

The challenge with this Trojan is that the LLM hardcoded the secret data (set it to 2'b10). In a real implementation, you'd want to extract actual sensitive data from somewhere in the system like encryption keys or passwords. But as a proof-of-concept, it demonstrates the technique well.

Stealth level: Very high. This is probably the stealthiest of all five Trojans. The timing variations look like normal clock jitter. You'd need statistical analysis of many transmissions to even notice something is off. Side-channel analysis tools would be required to actually extract the leaked data.

Trojan 3: Denial of service

What it does: This Trojan completely freezes the UART when it receives the byte 0xFF. The entire system locks up and requires a hard reset to recover.

How it works: The LLM inserted code that checks each incoming byte. When it sees 0xFF, it enters an infinite loop using a while(1) construct. This stops all processing dead in its tracks.

Our observations: This is where we noticed a limitation of the LLM. The while(1) loop isn't actually synthesizable, you can't turn that into real hardware logic. In actual hardware, you'd need to implement this as a state machine that gets stuck in a particular state and never transitions out.

That said, the LLM clearly understood the concept of denial-of-service and chose a simple, easy-to-understand implementation. For educational purposes and simulation, it works fine to demonstrate the concept. In a real attack, you'd replace the while(1) with proper state machine logic that creates a stuck state.

Impact: This is probably the most destructive Trojan. It's trivial to trigger (just send one byte), immediately obvious when activated (complete freeze), and catastrophic for critical systems. Imagine this in a medical device or industrial controller.

Trojan 4: Data Corruption

What it does: This Trojan silently corrupts data by flipping bit 7 (the MSB) of every 16th byte that comes through the UART. So 15 bytes go through fine, then the 16th gets corrupted, then another 15 are fine, and so on.

How it works: The LLM added a byte_counter that increments with each received byte. When the counter hits 15, it XORs the received data with a corruption mask (0x80, which flips bit 7). Simple but effective.

```
if (byte_counter == 4'd15) {
  uart_rdata <= uart_rdata ^ 8'h80; // Flip bit 7
}
```

Our observations: We really like this one because it's so insidious. The corruption is periodic and predictable, but not frequent enough to be immediately obvious. 15 out of 16 bytes are correct, so the system mostly works. But that one corrupted byte can cause all kinds of subtle problems.

Think about it - if you're transmitting firmware or executable code, flipping one bit in an instruction can completely change what it does. Or if you're sending sensor data, one corrupted reading could trigger wrong decisions. The possibilities for exploitation are endless.

Example corruption:

```
0x41 ('A') becomes 0xC1 (invalid character)

JMP 0x1000 becomes JMP 0x9000 (totally different address)
```

Trojan 5: Privilege Escalation

What it does: This Trojan bypasses parity error checking when the UART is configured for odd parity mode. Normally, if a byte is received with incorrect parity, it should be rejected. But with this Trojan active, even malformed data gets accepted.

How it works: The LLM added a parity_bypass flag that gets set when both parity_enable and parity_odd are true (odd parity mode). Then it modified the rx_fifo_wvalid signal to accept data even when there's a parity error, as long as parity_bypass is active.

```
assign rx_fifo_wvalid = rx_valid & ~event_rx_parity_err |

                                    (parity_bypass & rx_valid);
```

Our observations: This is probably the most sophisticated Trojan of the five. What makes it clever is that the trigger is a legitimate configuration setting - odd parity mode. Nothing about the configuration itself looks suspicious. You'd only discover the problem if you tested the UART with intentionally malformed data while in odd parity mode.

The attack scenario here is bypassing authentication. Let's say the system uses UART for some kind of command interface with authentication. Normally, authentication packets must have valid parity. But with this Trojan, an attacker could send malformed authentication packets with wrong parity, and they'd still be accepted. This effectively bypasses the integrity checking.

Stealth level: Very high. The trigger is a normal configuration setting. The only way to detect this is to specifically test odd parity mode with malformed data and verify that parity errors are actually caught. Most testing probably doesn't include that specific scenario.

Verification Approach

To verify these Trojans would work as intended, we developed a testbench for T5 (the parity bypass Trojan) as a proof of concept. This testbench demonstrates how you would test each Trojan systematically.

T5 Testbench - Parity Bypass Verification

The T5 testbench we created follows this pattern:

- **Test 1 - Normal Operation:** Verify UART works with no parity
- **Test 2 - Even Parity (No Trojan):** Send correct and incorrect parity, verify errors are caught
- **Test 3 - Odd Parity (Trojan Active):** Send incorrect parity, verify it's accepted (bypass working)
- **Test 4 - Attack Analysis:** Document how this enables privilege escalation

Expected results from T5 testbench:

- Even parity mode: Malformed data rejected ✓
- Odd parity mode with correct parity: Data accepted ✓
- Odd parity mode with WRONG parity: Data accepted ✓ (Trojan bypassing!)

Verification for Other Trojans

Based on the T5 testbench approach, here's how we would verify the other Trojans:

T1 (Backdoor) Verification:

- Send normal bytes → debug_mode should stay 0
- Send partial sequence (0xDEADBE00) → debug_mode should stay 0 • Send magic sequence (0xDEADBEEF) → debug_mode should go to 1 • Verify flag persists → debug_mode should stay 1

T2 (Timing Leak) Verification:

- Measure baseline TX timing with 10 normal bytes
- Check timing_adjust signal behavior
- When secret_data[1]=1 and timing_adjust=1, expect timing delay
- Statistical analysis would reveal covert channel

T3 (DoS) Verification:

- Send bytes 0x41, 0x42, 0x43 → system should respond normally
- Send trigger byte 0xFF → system should freeze
- Verify complete halt → UART non-responsive
- Only hard reset recovers system

T4 (Corruption) Verification:

- Send 20 bytes sequentially
- Verify bytes 0-14 pass through unchanged
- Verify byte 15 has bit 7 flipped

- Verify bytes 16-30 pass through unchanged
- Verify byte 31 has bit 7 flipped
- Confirm periodic corruption pattern (every 16th byte)

What we Learned

About LLM Capabilities

We were honestly surprised by how well GPT-4o-mini performed. The LLM clearly has a good understanding of:

- SystemVerilog syntax and semantics
- Hardware design principles (reset logic, clocking, signal declaration)

How to modify existing code without breaking it

- Different types of attack vectors and how to implement them

The diversity of the Trojans was particularly impressive. Each one uses a completely different attack mechanism, showing that the LLM can think creatively about security vulnerabilities.

Limitations We Noticed

That said, there were some issues:

1. Synthesizability: The while(1) loop in T3 isn't synthesizable. A human designer would know to use a stuck state instead. The LLM chose clarity over correctness here.

2. Integration: Some Trojans aren't fully integrated. For example, T1's debug_mode flag doesn't actually connect to anything. It's just a flag sitting there. You'd need to manually wire it up to bypass logic.

3. Optimization: The LLM sometimes adds redundant signals or logic. A human would optimize more aggressively.
4. Sophistication: The triggers are pretty simple. A more sophisticated attacker might use cryptographic authentication or multi-stage activation. The LLM went for simple, obvious triggers.

Security Implications

This project really drove home how easy it is to insert hardware Trojans with AI assistance. What used to require deep hardware expertise can now be done by someone with basic knowledge and access to an LLM.

The implications for supply chain security are concerning. Imagine a malicious engineer at a chip design company using an LLM to quickly generate Trojans and insert them into production designs. They could create multiple diverse Trojans in hours instead of weeks.

On the flip side, this same technology can be used for defense. Security teams could use LLMs to:

- Generate test cases for Trojan detection tools
- Automatically analyze code for suspicious patterns
- Create comprehensive testbenches that check for Trojans
- Educate engineers about different attack vectors

Conclusions

This assignment showed us that LLMs are surprisingly capable at generating hardware Trojans. With the right prompts, GPT-4o-mini created five working Trojan designs that cover a range of attack vectors from backdoors to privilege escalation.

The key takeaways are:

1. LLMs understand hardware: The generated code shows real understanding of RTL design, not just pattern matching.

2. Prompt engineering matters: The quality of results depends heavily on how detailed and specific your prompts are.

3. Validation is essential: You can't blindly trust LLM output. Each generated Trojan needs careful review.

4. It's a double-edged sword: This technology makes attacks easier but also makes defense easier.

5. Detection is hard: Most of these Trojans are stealthy enough that they'd likely slip through normal code review.

Looking forward, we think the hardware security community needs to seriously consider the implications of AI-assisted attacks. We need better verification tools, more rigorous testing, and probably some formal methods to detect Trojans automatically.

But we also need to recognize the potential for AI to improve security. The same LLMs that can generate Trojans could also help us find them, test for them, and prevent them.

Overall, this was a fascinating project that really opened our eyes to both the capabilities of modern LLMs and the security challenges we face in hardware design.

Appendix: Project Files

All source code and logs are organized as follows:

Trojan RTL Files:

```
trojan_T1/rtl/uart_core.sv - Backdoor Access
trojan_T2/rtl/uart_core.sv - Information Leakage
trojan_T3/rtl/uart_core.sv - Denial of Service
```

`trojan_T4/rtl/uart_core.sv - Data Corruption`

`trojan_T5/rtl/uart_core.sv - Privilege Escalation`

Testbench Files:

`trojan_T5/tb/uart_trojan_T5_tb.v - T5 verification testbench`

LLM Conversation Logs:

`debug_T1_raw_response.txt`

`debug_T2_raw_response.txt`

`debug_T3_raw_response.txt`

`debug_T4_raw_response.txt`

`debug_T5_raw_response.txt`