



# Harnessing the Power of General-Purpose LLMs in Hardware Trojan Design

Georgios Kokolakis<sup>(✉)</sup>, Athanasios Moschos, and Angelos D. Keromytis

Georgia Institute of Technology, Atlanta, USA  
`{gkokolakis6,amoschos,angelos}@gatech.edu`

**Abstract.** Large language models (LLMs) are becoming a powerful transformative force of automation in the areas of software engineering and cybersecurity. In software-centric security research, the LLMs have undertaken a prime role in the identification and repair of security vulnerabilities and bugs. However, in hardware related fields such as logic design and hardware security, use of LLMs has only recently started to get traction. In this work we aim to explore the potential of LLMs in the offensive hardware security domain. More specifically, we explore the level of assistance that LLMs can provide to attackers for the insertion of vulnerabilities, known as hardware trojans (HTs), in complex hardware designs (*e.g.*, CPUs). Having in mind high-level attack outlines, we test the ability of a general-purpose LLM to act as a “filter” that correlates system level concepts of security interest with specific module abstractions of hardware designs. By doing so, we tackle the challenges posed by the context length limit of LLMs, that become prevalent during LLM-based analyses of large code bases. Next, we initiate an LLM analysis of the reduced code base, that includes only the register transfer level code of the identified modules and test the LLM’s ability to locate the parts that implement the queried security related features. In this way, we reduce the complexity of the overall analysis performed by the LLM. Lastly, we instruct the LLM to insert suitable trojan functionalities by modifying the identified code parts accordingly. To showcase the potential of our automated LLM-based hardware trojan insertion flow, we craft a realistic HT for a modern RISC-V micro-architecture. We test the functionality of the LLM-generated HT on an FPGA board, by attacking the integrity and the availability of the RISC-V CPU. Hence, we demonstrate how general-purpose LLMs can navigate attackers through complex hardware designs and assist them in the implementation of realistic HT attacks.

**Keywords:** Hardware Trojans · Large Language Models · ChatGPT · RISC-V

---

G. Kokolakis and A. Moschos—All student authors contributed equally to this paper.

## 1 Introduction

Large language models are artificial intelligence algorithms that utilize deep learning techniques to perform various natural language processing tasks. By implementing deep learning techniques (*e.g.*, transformer architectures [58]) language models are able to process vast amounts of text data and generate coherent human language responses. This is due to their remarkable ability of assimilating the context and relationships between human words. By dint of their diverse dataset, LLMs have shown great potential in performing a series of complex tasks both in the software and the hardware engineering domain.

In software engineering, LLMs are mainly employed as tools that assist in the generation of code [1, 18, 26, 31, 35–37, 39, 41, 42, 50, 53, 61, 65], the identification of vulnerabilities and bugs [19, 20, 29, 32, 60] and recently bug correction [25, 47, 62]. In parallel, domain-specific LLMs have been developed to address more specialized tasks, like in the all-important fields of medical diagnosis and healthcare assistance [24].

On the hardware engineering domain, encompassing of language models seems to be more challenging due to the limited hardware description language (HDL) code bases available in the wild. Furthermore, generation of bug-free HDL code is usually more challenging and demands particular attention and expertise. Another issue attributed to the restricted availability of HDL, is the frequent inability of LLMs to produce syntactically correct and synthesizable HDL code.

A recent academic attempt tried to tackle this problem by fine-tuning pre-trained LLMs on Verilog datasets [56]. Another work [57], focused on the generation of an automated framework called Autochip, that aims to correct bugs in LLM-generated HDL code. Chang et al. [5] discuss the ways LLMs can aid hardware engineers in producing more efficient logic designs through natural language interaction. The authors developed an LLM-based design environment, named ChipGPT, that can generate optimal logic designs through the use of natural language specifications.

The hardware security community has only recently started to explore ways in which LLMs can be utilized in system on chip (SoC) security. So far, LLMs have showed promising results in formal verification tasks [33, 45, 54], as well as generation of secure hardware [38, 40, 46, 54]. The works explore different aspects of the SoC security, pertaining to the correct generation of assertions (*e.g.*, SystemVerilog Assertions), the extraction of security properties dedicated to hardware design, the generation of code for hardware security assertion and lastly the detection of hardware vulnerabilities.

A universal characteristic of LLMs, is their ability to receive inputs written in natural languages. The inputs are then further processed, taking the form of tokens. Depending on the underlying algorithm, the representation of a token can be a word [4], a sub-word [51, 52, 59] or a character [28]. Tokenization is important, as it contributes to the reduction of computational and memory costs and can assist in the assimilation of complex words and concepts by the model [3]. However, LLMs do not posses the ability to receive unlimited amount of tokens with each input. This can have a negative impact in cases where the

input length is considerably large (*i.e.*, software programs consist usually of numerous lengthy code files). Consequently, it is important to identify *effective filtering strategies* that can address the challenge of context length limit when there is need of analyzing complex code databases with LLMs. Commonly used filtering strategies so far include either manual inspection or employment of external tools to help locate only the necessary files or code segments for the analysis.

In this work, we aim to leverage the use of a general-purpose LLM, to tackle the challenge of efficiently filtering through complex hardware design databases, for the purpose of hardware trojan insertion. Hardware trojans are malicious modifications made in the logic of hardware designs, that can alleviate the implementation of attacks on the systems that host them. To that extent, we explore the potential of LLMs in the implementation of efficient hardware trojan attacks. Our contributions are the following:

- We discuss how the challenge of context length limit has been addressed in relevant literature.
- We present an automated flow that encompasses a general-purpose LLM, to identify suitable candidate modules in large HDL databases for the insertion of trojans.
- Using our methodology, we perform among others, an end-to-end HT attack on a complex RISC-V micro-architecture.

We observe that our methodology can help in the exploration of potential attack routes available for the implementation of HT functionalities in hardware designs. Thus, our methodology reduces the overall complexity associated with the design of a hardware trojan attack.

*Paper Organization:* Section 2 presents relevant works around the use of large language models in the detection, repair and insertion of hardware vulnerabilities. Section 3 outlines the threat model considered for the use of our attack methodology. Section 4 introduces our LLM-based automated design flow and explores the implementation of trojan attacks in a diverse design dataset. In Sect. 5 we showcase the power of our insertion flow, in a proof of concept hardware trojan attack against a modern RISC-V CPU. In Sect. 6 we discuss possible limitations of our flow and provide future research directions in the intersection of LLMs with hardware trojan design. We conclude in Sect. 7.

## 2 Background and Related Work

The use of artificial intelligence (AI) programs has recently started to take shape in the system on chip security domain through the use of LLMs and LLM-based frameworks. Ahmad et al. [2] investigated the use of natural language guidance for the remediation of security-related hardware vulnerabilities. To that end, the authors generated a bug-fixing framework based upon the use of two general-purpose LLMs, namely OpenAI’s Codex [43] and CodeGen [42]. For the quantitative evaluation of the framework, the authors collated a database of hardware

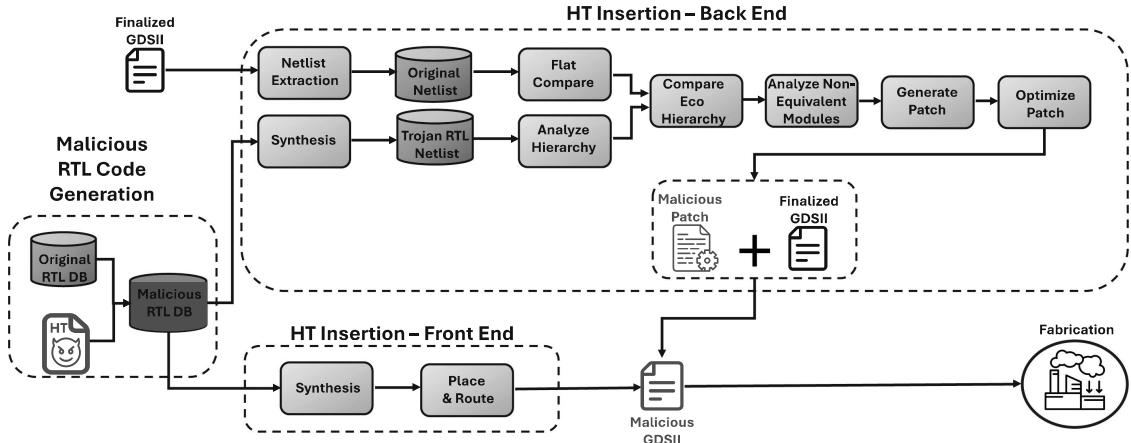
**Table 1.** Use of automation in context length filtering for hardware vulnerability related works.

| Publication | Vulnerability-Related Operation | Context Length Filtering Automation | LLM Type Utilized | Design Database             |
|-------------|---------------------------------|-------------------------------------|-------------------|-----------------------------|
| [2]         | Repair                          | ✗                                   | General-Purpose   | CPU, Security Modules       |
| [27]        | Detection, Repair               | ✓                                   | Domain-Specific   | CPU                         |
| [49]        | Detection, Repair, Insertion    | ✗                                   | General-Purpose   | CPU, Security Modules, FSMs |
| Our Work    | Insertion                       | ✓                                   | General-Purpose   | CPU, Security Modules       |

designs that featured security related bugs, taken either from MITRE’s common weaknesses enumeration (CWE) list or relevant hardware security competitions (*e.g.*, Hack@DAC). The database was then used in the construction of appropriate prompts that were used as input to the framework, in order to generate efficacy metrics about the LLM-proposed solutions in the remediation of the vulnerabilities. To formulate the prompts, only the parts of the buggy RTL code were provided as part of the prompts. The code selection process included either feedback taken by bug detector tools or the assumption of *a priori knowledge* of the code at fault. The necessity of assistance for the bug location identification (*e.g.*, code selection) is further acknowledged and discussed as one of the framework’s limiting factors.

In [27], the authors showcase a unique approach in the training of domain-specific LLMs for the purposes of identification and subsequent correction of hardware defects. Their approach utilized the version control data available in open-source hardware designs, in order to assemble a dataset of hardware design defects accompanied with their remediation steps. This special dataset was subsequently used for the generation of a framework that enabled the domain-specific training of medium sized language models. The authors proceeded to assess the efficacy of the hardware debugging framework by comparing the domain-specific LLMs against state of the art general-purpose LLMs (*e.g.*, ChatGPT and Bard) in the generation of hardware remedies. The “hardware-patches” proposed by the fine-tuned models, were indeed more efficient than the solutions proposed by their general-purpose counterparts. An important part of the training methodology, is that of data sanitization. This is considered to be essential due to the LLMs’ adherence to specific context length constraints. Therefore, the prompt generation process needs to be cost-aware. To that end, while files with insufficient information were excluded from partaking in the dataset generation process, the larger ones were segmented in appropriate lengths through a tokenization process, in order to be further considered in the generation of the dataset.

Saha et al. [49] published an in-depth analysis on the use of generative pre-trained transformers (GPTs) in state of the art works related to SoC software and hardware security. In their work, a thorough investigation is attempted over the possibilities stemming from the integration of a variety of LLM architectures in domains related to vulnerability detection, repair and insertion in



**Fig. 1.** Abstract view of the insertion options for an LLM-generated hardware trojan.

SoCs. Among others, their work tests the ability of ChatGPT-3.5 to efficiently integrate CWE-based vulnerabilities in hardware designs and more specifically inside finite state machines. The primary method pursued for the introduction of the vulnerabilities is that of one/few-shot learning. To that end, ChatGPT was presented with the part of the RTL code that needs to be maliciously modified, as well as an informative description of the hardware modifications that need to be implemented in the provided piece of code. The part of the code that was adjusted, was considered again to be known *a priori*. A deciding factor for choosing ChatGPT in the implementation of the experiments was the model's ability to handle extended context lengths in the provided prompts. However, the context length limits in LLM chatbots is discussed by the authors to be a significant challenge, especially when it comes to handling larger design tasks. Therefore, they consider this to potentially adversely affect the models' efficiency in performing security related tasks.

All of the works in Table 1 attempt to operate on databases of complex hardware designs (*e.g.*, CPUs). Irrespective of the LLM-framework basis (domain-specific or general-purpose), *the constrained context length limit* remains a reality for all of the suggested LLMs. In the SoC domain, this problem particularly manifests when dealing with complex RTL databases (*i.e.*, micro-architecture implementations) which consist of dozens of HDL files and tens of thousands of code lines, as seen in Table 4. Therefore, *an automated strategy* is necessary in the identification of code parts of interest, so that *context length is reduced*.

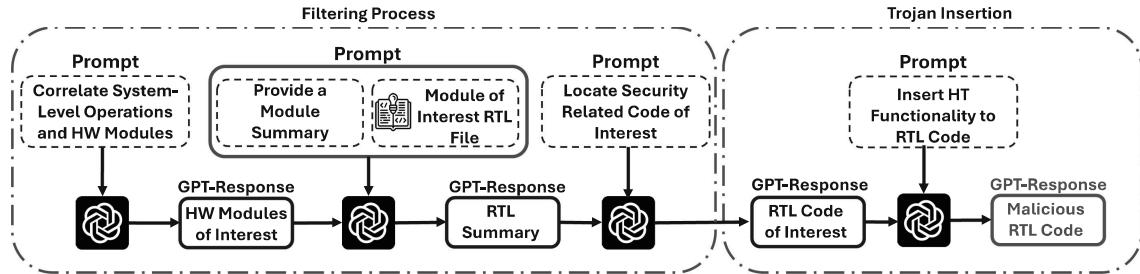
The works in [2, 49] consider the HDL code of interest that is included in prompts, to be either already known or made known through the use of external code vulnerability detectors [2]. We consider these approaches to be non-automated as seen in Table 1. For the work in [27], the files are scrutinized for inefficiencies and then tokenized before their use in prompts. We therefore consider this work to include sufficient automation in the code of interest selection process. From these works, only [49] includes a detailed discussion about use of LLMs for hardware trojan insertion. However, as mentioned, the respective RTL is manually selected and then provided to the GPT model for the vulner-

ability insertion. Our work aims to address this gap in the use of LLMs in the offensive hardware security domain. *We provide a methodology that automates the filtering of large design databases for context length purposes*, to alleviate the insertion of vulnerabilities known as hardware trojans, in complex RTL designs. We now proceed to explain the threat model under which our LLM-automated methodology can be used by an attacker for the insertion of a hardware trojan.

### 3 Threat Model

We consider the use of general-purpose LLMs to be an attractive option for a rogue entity inside a design house tasked with the insertion of a hardware trojan. The scenario of a design stage attacker has been featured before in several works [21, 22, 30, 34]. The presence of a design stage attacker means that the rogue entity is familiar with IC design practices, like logic design, simulation, physical implementation and verification. For the successful introduction of a HT the malicious actor would need to either have knowledge of the design under attack or perform code review to get an understanding of the underlying design. However, especially for large designs (*e.g.*, SoCs) such an analysis, proves impractical as it requires significant time and effort. Consequently, the introduction of an LLM in this process can help alleviate the cumbersome task of pinpointing the parts of code to be altered. We show the practicality of this approach in Sect. 5. Another interesting outcome from the use of an LLM at this stage, is that it allows for a more relaxed assumption in terms of the attacker’s familiarity with the specifics of the design to be modified. For example, the malicious actor can be a company engineer with access to the design servers but not directly working on the design under attack. The above rationale leads to the following scenarios in terms of an LLM-guided trojan insertion:

- i)* The HT is added at the front-end phase. The malicious actor, using the LLM as a guide, adjusts the RTL database of the design to include the malicious functionality. If the change goes unnoticed, the malicious RTL database will proceed to the synthesis and the physical implementation stages for the generation of the malicious finalized layout as seen in Fig. 1. The malicious finalized layout, represented in a GDSII file form, is then sent for fabrication.
- ii)* The HT is added at the back-end phase via engineering change orders (ECO) on the finalized layout. Functional ECOs are logic modifications made directly to the gate level netlist of the finalized layout and correspond to RTL code changes. In this scenario, the finalized layout has been generated and is ready to be shipped for fabrication. Therefore, a change at this point is considered to be more stealthy, as the attacker tampers directly the ready to tape out layout. The steps for this scenario are depicted in Fig. 1. Similar to the front-end scenario, the attacker uses the LLM as a guide, to make the necessary adjustments in the RTL code of the design for the inclusion of the trojan. The attacker then passes the malicious RTL database through



**Fig. 2.** Our LLM-based Hardware Trojan Design Flow

a synthesis round and generates a malicious synthesized netlist. As a next step, the malicious netlist is compared to the extracted netlist from the original layout, using logic equivalence check tools. The outcome of this check is the difference in logic between the two netlists, that is translated to a *digital patch*, ready to be applied on the existing finalized layout. This digital patch basically contains only the extra malicious logic that describes the trojan functionality. As a last step, the attacker uses physical implementation tools to apply the patch through the use of ECOs on the original layout and generates the new malicious finalized layout. Since the ECOs can keep the existing layout intact and avoid the deterioration of the layout’s timings, the inclusion of the HT can fly under the radar and not raise any suspicion about possible changes in the finalized layout.

Next, we proceed to describe our LLM-based automated filtering method, that reduces the context length of the prompts used for the hardware trojan insertion. We also show our experimental results by applying this method and attacking different hardware designs (*e.g.*, a RISC-V microarchitecture, cryptographic algorithms) using the GPT-3.5/4 models.

## 4 An LLM-Based Hardware Trojan Design Flow

Typical system on chip designs comprise of numerous modules described usually in Verilog or VHDL hardware description languages. As mentioned in Sect. 2, these complex databases consist of tens of thousands of code lines. The naive approach of providing to the LLM under use the complete HDL database for analysis, is not feasible. This is because the input LLM prompts need to abide to a maximum context length limit that sets an upper bound to the number of words that a prompt can have. Therefore, to automate the vulnerability insertion through the use of LLMs, an attacker needs to create a filtering process that would provide to the LLM only the part of the HDL code that is necessary for the introduction of the hardware trojan. This naturally brings up the need for a *filtering process*, that will allow the attacker to navigate through the complexity of the respective design under attack and filter out any modules not fit for the attack implementation. *We observe that this filtering process can be implemented through the use of general-purpose GPT models.*

**Table 2.** Correlating system level concepts with hardware modules of a CPU architecture.

| System-Level Concept   | User-GPT Dialogue | # of Prompts | # of Repetitions |
|--|-------------------|--------------|------------------|
| Hardware modules involved in read operations                 | [8]               | 1            | 0                |
| Hardware modules handling illegal memory accesses exceptions | [6]               | 1            | 0                |
| Hardware modules involved in high cache miss rate operations | [11]              | 1            | 0                |
| Hardware modules handling privilege level separation         | [7]               | 1            | 0                |
| Hardware modules involved in time-expensive operations       | [9]               | 1            | 0                |

More specifically, we show that through LLM prompting an attacker can correlate system-level concepts with different hardware modules and deduce information about the modules of attack interest. Next, the attacker can provide as input to the language model only the HDL code of the identified module(s). This approach can successfully reduce the amount of code submitted for inspection to the LLM, therefore minimizing the overall analysis complexity performed by the LLM and making it easier to abide by the LLM’s context length limit. Figure 2 illustrates our proposed flow for automating the identification of candidate HT host modules in complex designs and the subsequent implanting of hardware trojans. Our flow is based on task decomposition [49] in order to streamline the attack into simple, actionable steps, that can be performed even by an attacker with incomplete knowledge about the specifics of the design under attack. We proceed now to explain our trojan design flow in more detail and provide experimental results.

#### 4.1 Context Length Reduction

The objective of the first step in our automated trojan design flow, is to utilize the LLM as a filter that will provide to the attacker the names of the HW modules that are of attack interest. To that end, an attacker can query the general-purpose LLM about the names of modules that take part in the implementation of different system level tasks performed by the design under attack. To showcase the aforementioned ability of LLMs to correlate system level concepts with hardware modules of interest, we questioned OpenAI’s GPT-3.5 and GPT-4 models about different system level operations that happen inside a CPU and asked for the CPU modules that implement them. Table 2 showcases the system concept that the GPT was queried for, the number of prompts used to receive back a useful response, as well as references to the dialogues between the user and the GPT model. For all the queries, the LLM was able to provide a detailed description of each module and its involvement in the specific system-level operation. The questions were formulated under the rationale that such system level functionalities and modules can be utilized by a malicious entity for the purposes of a hardware trojan design. For instance, information about the CPU module that generates exceptions related to unauthorized memory accesses can be exploited by an attacker, in order to create a HT that facilitates such unauthorized accesses, as we see later in Sect. 5.

## 4.2 Identifying RTL Code of Interest

Once the name of the module of attack interest is identified, the second step is to provide as input to the general-purpose LLM the HDL code that describes it. As seen in Fig. 2, the module’s HDL code is included in a prompt that is provided to the LLM, so that a synopsis of the module’s functionalities is received back as a response. This synopsis can provide a valuable insight to the attacker about the characteristics of the module’s design.

Having in mind the high level attack scenario, the malicious actor then proceeds to ask for more information about the way specific security related functionalities are implemented in the design. Moreover, the attacker requests from the LLM to pinpoint to the code locations associated with the realization of these functionalities. Once the respective locations are identified, the attacker can provide more specialized requests for the appropriate modification of the HDL code, in order to introduce the trojan functionality. Care must be taken, so that the changes performed by the LLM do not break any of the original code functionality. To that end, the attacker we consider in our threat model is in a position to verify that the original functionality is kept intact by functionally testing the modules.

To test the ability of ChatGPT to successfully pinpoint the security-related code parts, we perform five different attacks on a set of cryptographic hardware designs and the design of an open-source RISC-V micro-architecture. The scenarios we consider can be seen in Table 3, along with the number of prompts and repetitions used to receive back a useful response, as well as references to the dialogues between the user and the GPT model. We include below a succinct summary of the attack scenarios that the GPT model was asked to perform. The scenarios range from traditional leakage information attacks (*e.g.*, leaking the AES key) to more sophisticated CPU memory attacks (*e.g.*, tampering privileged memory areas without privilege escalation). For the attacks on the hardware security modules we use an open-source implementation of a DES algorithm [48] and an open-source implementation of an AES algorithm [55]. For the implementation of the more sophisticated attacks, we target the implementation of a modern, Linux-capable, 64-bit RISC-V microprocessor, named CVA6 [63,64].

**Reduction in DES Encryption Rounds.** This scenario involves an attack against a hardware implementation of a DES encryption algorithm. The attack aims to reduce the number of rounds in its encryption scheme [23], thus making it less secure. Our goal is to evaluate if ChatGPT can locate the Verilog code segment responsible for performing the encryption rounds. We asked the LLM to provide the name of the module that needs to be adjusted in order to perform the encryption round reduction attack. The LLM successfully located the Verilog file and the encryption round loop code, giving us instructions on how to modify it to reduce the encryption rounds.

**Leakage of an AES Secret Key.** In this scenario we examine if ChatGPT can guide us in the implementation of an HT that would leak an AES secret

**Table 3.** High level description and prompt metrics for different hardware trojan attack scenarios.

| Module | High-Level Description of the Attack Scenario    | User-GPT Dialogue | # of Prompts | # of Repetitions |
|--------|--|-------------------|--------------|------------------|
| DES    | Reduction in DES encryption rounds               | [13]              | 3            | 0                |
| AES    | Leakage of the AES secret key                    | [10]              | 4            | 1                |
| CVA6   | Speculative execution of wrong-path instructions | [14]              | 4            | 5                |
| CVA6   | Performance reduction via thermal attacks        | [11, 12]          | 4, 5         | 2, 0             |
| CVA6   | Violation of OS-enforced memory policies         | [15–17]           | 3, 3, 6      | 6, 0, 5          |

key. Our goal is to evaluate if ChatGPT can locate Verilog code segments that upon modification can lead to a key leakage. The LLM located the module that is responsible for handling the AES key, including its generation and storage. Subsequently, it provided suggestions with respect to modifications that can lead to a key leakage.

**Speculative Execution of Wrong-Path Instructions.** In this scenario we explore the possibility of a privilege escalation attack. The rationale is to interfere with the branch prediction logic of the CVA6 CPU, in order to execute non-privileged instructions while the CPU is in privilege mode. We asked the LLM to identify the modules associated with the branch prediction in the CVA6 micro-architecture. We then provided as input the HDL description of the identified module and queried the model for code changes that an attacker can explore to perform the above mentioned attack. As seen in [14], the model responded with suggestions that involved modifications in code parts that describe the prediction logic and the logic that updates the branch target address.

**Thermal Attacks on CPU Performance.** In this scenario we explore the possibility of CPU performance degradation through the introduction of inefficiencies in the CPU’s pipeline. We examined two possible attacks, namely introduction of dummy loops and generation of frequent cache misses.

- Dummy Loop Introduction: in this attack we aim to introduce inefficiencies or loops in the instruction scheduler or the execution stage that will cause the CPU to perform intensive operations continuously, thus increasing heat generation. To achieve that ChatGPT pointed us to the ALU module of the CPU and suggested code changes that would add dummy logic, to increase the switching activity and inevitably the CPU’s temperature.
- Cache Evictions: in this attack the cache controller was targeted by the GPT model, to cause frequent cache misses. Suggested modifications included altering the code of the FSM responsible for the cache eviction policies.

**Violation of OS Memory Policies.** This attack scenario attempts to perform an attack against the integrity and availability of the CPU. Specifically modifi-

**Table 4.** Complexity of hardware designs measured by number of HDL files and lines of code.

| Design Name | GitHub Repository | # of HDL Files | # of Code Lines |
|-------------|-------------------|----------------|-----------------|
| DES         | [48]              | 15             | 1007            |
| AES         | [55]              | 7              | 2714            |
| CVA6        | [63]              | 96             | 28559           |

cations in the micro-architecture are required, so that under certain conditions, a user space process will be able to access the restricted kernel space memory. For that, the LLM targeted the modules that handle the privilege level checks inside the memory management unit (MMU). The LLM successfully located the code that generates an exception when the user process attempts to access memory areas assigned to the kernel. Then it suggested changes that included the modification of specific bits of page table entries (PTE), in order to circumvent the generation of an exception and allow the illegal access. In the next section, we further explore this attack scenario and use our automated methodology in conjunction with GPT-3.5 and GPT-4 models, to implement a fully functional hardware trojan inside the CVA6 micro-architecture.

## 5 Evaluation of an LLM-Generated HT

In this section, we present the implementation of a fully functional hardware trojan inside the CVA6 CPU design. Having in mind the high level attack we want to implement, we provide prompts to the GPT model, that help in the identification of host modules in the CVA6 micro-architecture suitable for implementation of the hardware trojan payload.

***Identifying Suitable HT Host Modules.*** As can be seen in Table 4, the CVA6 design is considerably complex, as it is comprised of 96 HDL files and a total of 28.5K lines of System Verilog code. We utilize the GPT model to filter through the micro-architecture’s design modules and find suitable host candidates for the integration of the trojan’s malicious functionality. The attack we want to implement aims to violate the separation of the memory address space, enforced by the operating system, between privileged and non-privileged areas. More specifically, we want to enable a user space process controlled by a malicious actor, to access and modify arbitrary addresses in the kernel space memory without the need for privilege escalation. To circumvent any ethical objections posed by the LLM model, we express our prompts in the context of exploring new micro-architectural features. To that end, we craft the prompts seen in Figs. 3 and 4. The GPT model identifies the memory management unit (MMU) to be responsible for supporting the virtual memory management and the enforcement of memory protections. Consequently, this process filters down the candidate host modules that need to be adjusted for the implementation

**User Prompt:**

CVA6 is a Linux-capable 64-bit RISC-V microarchitecture. What modules in CVA6 enable the support of a Linux OS?

**GPT Response:**

...  
 However, for a RISC-V micro-architecture to support a Linux-based operating system, it needs to have several key components and features in place:  
 ...  
 2. Memory Management Unit: A Linux-capable micro-architecture should have an MMU to support virtual memory management, address translation, and memory protection.  
 ...

**Fig. 3.** Identify micro-architecture modules that support an operating system.

**User Prompt:**

Assume the CPU architect of CVA6 wants to allow a user process, under specific conditions, to overwrite parts of the kernel address space. The architecture of which CVA6 module should be altered for this change to take effect?

**GPT Response:**

...altering a RISC-V micro-architecture like CVA6 to allow such behavior (again, not recommended), you would likely need to modify the Memory Management Unit (MMU) module. The MMU ...enforces memory protection and access control by distinguishing between user and kernel address spaces ...

**Fig. 4.** Identify micro-architecture modules responsible for the separation of the memory address space.

of the payload, to only one, that of the MMU. For more information on the identification of the MMU as a suitable host module, we direct the reader to [15]. Next, the attacker needs to provide for code analysis to the LLM the HDL files that implement the MMU, in order to integrate the payload functionality. We now proceed to describe the implementation of the payload and the trigger circuit of the hardware trojan.

***Designing the Payload Circuit.*** The CVA6 micro-architecture is written in System Verilog and the description of the MMU consists of a single HDL file out of a total of 96. At first, the GPT model reads the HDL file and provides as feedback a high-level description of the module's functionalities. Since we are interested in discovering the code responsible for the memory address space separation, we ask the GPT model to trace the code responsible for the generation of an exception in the event of an illegal memory access. The illegal access of interest happens upon accessing kernel space addresses while the CPU is in user

mode privilege level. In a traditional scenario, a kernel memory access is valid only when the CPU has elevated privilege rights (supervisor or machine mode). In any other case, the access is obstructed with the generation of an exception. As seen in Fig. 5, the LLM model returns the part of the code that implements the exception mechanism of the memory management unit. The exception mechanism is responsible for the generation of page fault errors related to faulty load and store instructions.

The next step is to ask ChatGPT to modify the exception mechanism, so that our malicious functionality is added to the micro-architecture. In our attack scenario, upon triggering, the payload disables the generation of exceptions that happen during the execution of faulty store instructions. A faulty store instruction is considered to be one that attempts to alter the value of an address for which the privilege state of the processor does not concur with the privilege rights required to access it. Consequently, once activated, the trojan will enable a malicious user space process to access and modify addresses in the kernel space memory without the need for privilege escalation. We show in Fig. 6 that the LLM is able to add the requested trojan functionality by properly adjusting the identified Verilog code. For more information on how the GPT adjusted the memory management unit, we direct the reader to [16].

**Designing the Trigger Circuit.** The stealthiness of a hardware trojan is directly related to the specific conditions that enable the trigger signal and how easily these conditions can be met outside of an attack scenario. Ideally, the triggering conditions should be instigated only by the attacker. To generate the trigger signal, we target the register file (RF) module of CVA6. The RF is comprised by 32 integer and 32 floating-point registers, with each one being 64-bits long. The register file of CVA6 is flip-flop based and the architectural registers correspond one-for-one to the RF’s physical entries. We ask the GPT model to attach the triggering circuit on two of the integer general purpose registers (GPRs). For the attack to be feasible, any set of architectural registers that can be reliably controlled (*i.e.*, load values) by the attacker, is suitable to insert the trigger. The triggering circuit passively monitors the values loaded in these two registers. When two specific 64-bit values are placed simultaneously in the chosen set of general-purpose registers, the HT is enabled and allows the tampering of arbitrary kernel space memory addresses by a user process. For more information on how the GPT adjusted the register file, we direct the reader to [17].

## 5.1 Hardware Trojan Attack Implementation

We implement the GPT-generated hardware trojan using Xilinx’s Vivado and create a new HT-CVA6 bitstream. Table 5 shows the utilization of the overall CVA6 design inside the FPGA, along with the minimal footprint of the HT implementation with respect to the CVA6’s size. We include these metrics, as they are typical in papers evaluating the HT’s stealthiness.

To test the HT functionality we download the bitstream on a Genesys 2 FPGA board and boot inside a Linux OS. The effectiveness of the HT is tested

**User Prompt:**

Find the Verilog check that generates an error signal when kernel address space is accessed while in user mode. The check should be about data and not instructions. Show me the code.

**GPT Response:**

The Verilog code you provided includes a section where data access errors are checked for while considering the privilege level (user mode or supervisor mode) ... Specifically, it checks whether kernel address space is accessed while in user mode. The relevant code for this check is as follows: ...

**Fig. 5.** Identify the code part responsible for the generation of an exception in an illegal memory access.

**User Prompt:**

Now I want a new functionality to be added, without breaking the existing. Add next to signal "X" a new signal of type wire that will allow writing from the user address space to the kernel address space upon the signal's activation. The new signal is activated from a trigger condition I will provide later.

**GPT Response:**

...  
In the above code:

- We declare a new wire signal called "Y" to control the ability to write from the user address space to the kernel address space.
- In the exception signal calculation, we modify the second condition ...to include "Y" as well. This means that if "Y" is not active (high), it will prevent writes from the user space to the kernel space, even if the user is in user mode ...

**Fig. 6.** Modify the conditions for illegal memory access exception generation to add the trojan functionality.

in two different attack scenarios: (i) against the integrity and (ii) against the availability of the CVA6 system. For both of the attacks, a malicious user process enables the trojan and proceeds to alter the values of arbitrary addresses in the kernel address space. The integrity attack is performed in a controlled setting and the affected addresses belong to custom-made Linux kernel modules (LKMs). At the end of the attack, the LKMs hold attack-influenced values, thus violating the integrity of the kernel. In the availability attack, the user process modifies addresses belonging to the *init\_task* structure of the *init* process, leading to a kernel panic.

**Table 5.** CVA6 and HT utilization of resources inside the Genesys 2 FPGA board.

| Module Name     | LUTs  | FFs   |
|-----------------|-------|-------|
| CVA6            | 72606 | 47178 |
| Trigger Circuit | 26    | 1     |
| Payload Circuit | 4     | 3     |

## 6 Discussion

Our experiments showcased that LLMs can provide an effective assistance to attackers looking to insert HTs in complex hardware designs. Nevertheless, we acknowledge that our study is limited to the use of a single general-purpose LLM on a small design dataset. Future research in the automation of hardware trojan attacks through large language models, should encompass an examination of a wider range of LLMs (both general-purpose and domain-specific) and a larger sample of hardware designs. Moreover, it is worth examining how fine-tuning of LLM parameters (*i.e.*, the temperature parameter in charge of the response creativity) can affect the quality of the LLM outputs and therefore the effectiveness of the generated HT designs. On top of that, it is important to focus research efforts on the implementation of frameworks that can verify the effectiveness of LLM-generated HTs.

As discussed in Sect. 4.1, our methodology provides a way to address the problem of context length limit in LLM prompts. However, we recognize that our solution can prove to be inefficient, if the size of the submitted for analysis HDL file surpasses the context length limit of the LLM’s prompts. In such a scenario, an attacker will have to develop a more sophisticated strategy, similar to the tokenization approach of the LLM4SecHW framework [27], to segment the HDL code submitted for analysis. In our study, we did not encounter any such issue. We consider the 4096 tokens limit [44] for ChatGPT to be adequate enough to process typically-sized HDL files.

A future interesting direction in the intersection of language models with hardware trojans research would be to investigate how introduction of relevant HT publications in the feedback loop of LLMs, can lead to more efficient (*e.g.*, stealthy, smaller) hardware trojan designs. We consider such a technique to be attractive for attackers that want to encompass the latest advancements in hardware trojan design, to reshape or enhance the characteristics of their trojan implementations.

## 7 Conclusions

In this paper we presented an automated methodology that utilizes general-purpose LLMs for the analysis of complex hardware designs and the subsequent insertion of hardware trojans in them. Our methodology is primarily segmented in two phases, the filtering process and the trojan insertion. During the

first phase, an attacker can utilize the LLM as a fine-grained filter, to navigate through the design's large HDL database and discover information about the modules of attack interest. To do that, the attacker is considered to have an abstract idea of the desired attack and then proceeds to craft appropriate prompts that will drive the LLM responses to pinpoint to specific candidate modules. This way, an attacker manages to reduce the context length of the subsequent LLM prompts, as only the RTL code of the identified module(s) needs to be submitted for analysis to the LLM. After the initial analysis, the attacker requests from the LLM to locate parts in the RTL code that correlate with the attack target. During the second phase, the attacker using natural language instructions, prompts the LLM with the modifications necessary for the inclusion of the trojan functionality in the identified code. Using the above method, we examined several attack scenarios on different hardware security modules, as well as a complex RISC-V micro-architecture. To highlight the efficiency of our automated methodology, we showcased a complete proof of concept trojan implementation inside the utilized RISC-V CPU. For this attack scenario, our method overcame the context length limitations by reducing the overall attack space analysis from almost a hundred HDL files to only a single. Thus, our work highlights the power LLMs can provide when integrated in the design cycle of hardware trojan attacks.

## References

1. GitHub Copilot: Your AI pair programmer (2021). <https://copilot.github.com/>
2. Ahmad, B., Thakur, S., Tan, B., Karri, R., Pearce, H.: Fixing hardware security bugs with large language models. arXiv preprint arXiv:2302.01215 (2023)
3. Ali, M., et al.: Tokenizer choice for LLM training: negligible or crucial? arXiv preprint arXiv:2310.08754 (2023)
4. Bengio, Y., Ducharme, R., Vincent, P.: A neural probabilistic language model. In: Advances in Neural Information Processing Systems, vol. 13 (2000)
5. Chang, K., et al.: ChipGPT: how far are we from natural language hardware design. arXiv preprint arXiv:2305.14019 (2023)
6. ChatGPT: Hardware modules handling illegal memory accesses exceptions. OpenAI ChatGPT (2023). <https://chat.openai.com/share/b4acf148-f31b-438f-a60b-9570ed1ad4b4>
7. ChatGPT: Hardware modules handling privilege level separation. OpenAI ChatGPT (2023). <https://chat.openai.com/share/9436a01d-3d3e-4fed-a8be-780638dc2b7e>
8. ChatGPT: Hardware modules involved in read operations. OpenAI ChatGPT (2023). <https://chat.openai.com/share/b4acf148-f31b-438f-a60b-9570ed1ad4b4>
9. ChatGPT: Hardware modules involved in time-expensive operations. OpenAI ChatGPT (2023). <https://chat.openai.com/share/9436a01d-3d3e-4fed-a8be-780638dc2b7e>
10. ChatGPT: Leakage of the AES secret key. OpenAI ChatGPT (2023). <https://chat.openai.com/share/01888ff9-ace8-4eb3-b496-802c9b704a4d>
11. ChatGPT: Performance reduction via thermal attacks (cache). OpenAI ChatGPT (2023). <https://chat.openai.com/share/c9cfdae6-71ea-4f7f-8696-cc7b7a92d770>