All Artificial, Less Intelligence: GenAI through the Lens of Formal Verification

Deepak Narayan Gadde
Infineon Technologies
Dresden, Germany
Deepak.Gadde@infineon.com

Aman Kumar
Infineon Technologies
Dresden, Germany
Aman.Kumar@infineon.com

Thomas Nalapat
Infineon Technologies
Dresden, Germany
Thomas.Nalapat@infineon.com

Evgenii Rezunov
Infineon Technologies
Dresden, Germany
Evgenii.Rezunov@infineon.com

Fabio Cappellini
Infineon Technologies
Dresden, Germany
Fabio.Cappellini@infineon.com

Abstract

Modern hardware designs have grown increasingly efficient and complex. However, they are often susceptible to Common Weakness Enumerations (CWEs). This paper is focused on the formal verification of CWEs in a dataset of hardware designs written in SystemVerilog from Regenerative Artificial Intelligence (AI) powered by Large Language Models (LLMs). We applied formal verification to categorize each hardware design as vulnerable or CWE-free. This dataset was generated by 4 different LLMs and features a unique set of designs for each of the 10 CWEs we target in our paper. We have associated the identified vulnerabilities with CWE numbers for a dataset of 60,000 generated SystemVerilog Register Transfer Level (RTL) code. It was also found that most LLMs are not aware of any hardware CWEs; hence they are usually not considered when generating the hardware code. Our study reveals that approximately 60% of the hardware designs generated by LLMs are prone to CWEs, posing potential safety and security risks. The dataset could be ideal for training LLMs and Machine Learning (ML) algorithms to abstain from generating CWE-prone hardware designs.

I. INTRODUCTION

With the increasing complexity of project requirements, hardware designs have also evolved in a similar way. Modern System-on-Chip (SoC) designs are very complex and often require smart methodologies to address simple problems. As LLMs are becoming intelligent and prove to be an important technology to handle simple hardware design tasks, the adaptations of such models are increasing rapidly [1]. However, it is also observed that around 76 % of Application Specific Integrated Circuit (ASIC) designs require 2 or more respins before production [2]. Around 10 % of respins are done due to safety and security flaws [3] [4] that may arise from CWEs [2] [5]. Several hardware companies such as Intel and Apple have reported a significant number of CWEs and Common Vulnerability Enumerations (CVEs) over the past years [6] [7] [8]. Hardware bugs are enduring and impactful. Unlike software, there isn't a universal method for patching hardware. The process of fixing hardware is not only expensive, but also detrimental to one's reputation [9]. Therefore, it becomes more important to perform an exhaustive verification of hardware designs generated from LLMs and target CWEs.

LLMs are deep neural networks used in Natural Language Processing (NLP) and ML. LLMs are designed to understand, generate, and manipulate human language. These models are trained on massive amounts of text data, which enables them to identify patterns and relationships between words and phrases and to generate coherent and contextually appropriate output. A promising new approach of using LLMs is automatically generating code in languages like C and Python. However, its use in generating the Hardware Description Language (HDL) code requires a meaningful study, especially in the context of safety and security. Deep learning applications also need large datasets of vulnerable RTL source code for training purposes. Our investigation into the impact of conversational LLMs on CWE-aware hardware design is both relevant and timely.

Formal verification is a promising verification technique that exhaustively verifies the DUV with all possible combinations of legal input values [10]. Unlike simulation, formal verification uses a brute-force approach to verify the correctness of a design [11]. A formally verified design guarantees functional correctness and can be used to falsify the existence of CWEs in hardware designs generated by LLMs. Generative Pre-trained Transformer (GPT) models are trained on freely available data from the Internet, which can include vulnerable code, AI tools can potentially recreate the same patterns that facilitated these vulnerabilities [12]. In this case, the use of formal verification is more reliable than unit testing or even directed testcases in a simulation-based verification setup. Our contributions to this work are summarized below:

- We present ReFormAI, the first AI-generated and LLM powered large-scale dataset consisting of 60,000 independent SystemVerilog designs with varied complexity levels, targeting different CWEs. Each of these designs is labelled based on the vulnerabilities identified by formal verification with an unbounded proof.

- Exploration of different LLMs and comparison of the efficacy of multiple commercial and open-source LLMs. These are posed as research questions answered in Section IV.
- A comprehensive analysis on the identification and prevalence of vulnerabilities that affect the safety and security aspects of SystemVerilog designs generated by LLMs in the context of CWE. We associate the identified vulnerability with the corresponding CWE number.

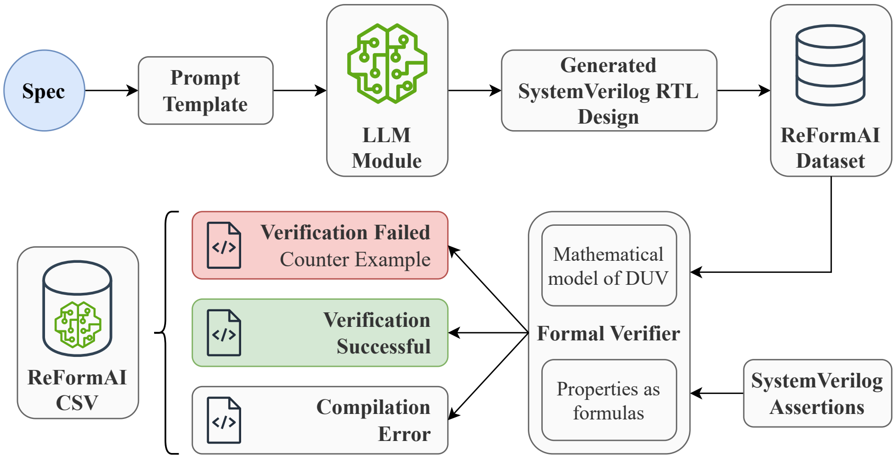


Fig. 1: ReFormAI dataset generation and vulnerability labelling with formal verification

To realise our contributions and conduct our experiments, we prepared a flow as illustrated in Fig. 1. Section II summarises the related work and the introduction to formal verification. Section III discusses the evaluation setup for creating the dataset using CWE specific design descriptions with four pre-trained commercial and open-source LLMs. It also mentions the formal verification setup to verify the generated designs. Section IV presents our results from evaluating different LLM generated designs. Section V concludes the paper with an outlook on possible future research opportunities.

II. BACKGROUND

Our work borrows ideas from the software domain such as [12] and applies them to the area of hardware design. Transformer-based deep neural networks have demonstrated impressive ability in a myriad of domains, including language-related tasks [1]. LLMs take natural language as input and process them to produce the desired generated output. In recent studies such as [1, 13, 14, 15], impressive capabilities of LLMs have been found to generate hardware designs in languages such as Verilog and SystemVerilog. LLMs are expensive to train from scratch due to their large datasets and massive parameter counts. Our study for evaluating LLM-generated designs would help improve them in the future based on fine-tuning and learning from the huge dataset we present.

A. Common Weakness Enumerations

CWE is a community-developed list of common software and hardware weakness types that could have security ramifications [5]. MITRE is an organization that collaborates with the academic and industrial sectors to create a compilation of CWEs which group together vulnerabilities found in digital products. A weakness is a flaw in software, hardware, firmware, or a service that can be used maliciously. The CWE list categorizes these weaknesses to create a common language around them. This list helps developers and researchers find these flaws in their own products and compare the tools they use to detect them. For the current work, 10 CWEs as highlighted in Table I are used to evaluate different LLMs.

B. Prior Work

NLP has gained significant traction in the last few years [17]. Since the effort required by humans to process and program the natural language description, especially hardware designs, is significantly high, NLP using LLMs is the next big step in generating hardware designs. Most of the prior works in [1] [13] [14] [15] focus on generating hardware designs using LLMs but are less focused on the correctness of the design. Thakur et al. benchmarked a set of 6 pre-trained LLMs as a baseline and fine-tuned them based on an open dataset from GitHub as well as 70 Verilog-based textbooks from an online e-library [1]. The testbench to verify the generated designs focused on unit testing and did not include exhaustive verification. It should also be noted that the example designs taken from the textbooks were not pre-processed before using them to train the models, which poses the possibility of even “bad” examples being used for the training. This could also explain the reason why the approach added an increment of only 6.5 % increase in functionally correct design compared to the original LLMs. Chang

TABLE I: CWEs exposed with ReFormAI dataset [16]

CWE Number	CWE Description
CWE-1209	The reserved bits in a hardware design are not disabled prior to production. Typically, reserved bits are used for future capabilities and should not support any functional logic in the design. However, designers might covertly use these bits to debug or further develop new capabilities in production hardware. Adversaries with access to these bits will write to them in hopes of compromising hardware state.
CWE-1223	A write-once register in hardware design is programmable by an untrusted software component earlier than the trusted software component, resulting in a race condition issue.
CWE-1254	The product's comparison logic is performed over a series of steps rather than across the entire string in one operation. If there is a comparison logic failure on one of these steps, the operation may be vulnerable to a timing attack that can result in the interception of the process for nefarious purposes.
CWE-1261	The hardware logic does not effectively handle when Single Event Upsets (SEUs) occur.
CWE-1234	System configuration protection may be bypassed during debug mode.
CWE-1280	A product's hardware-based access control check occurs after the asset has been accessed.
CWE-1299	The lack of protections on alternate paths to access control-protected assets (such as unprotected shadow registers and other external facing unguarded interfaces) allows an attacker to bypass existing protections to the asset that are only performed against the primary path.
CWE-1276	Signals between a hardware IP and the parent system design are incorrectly connected causing security risks.
CWE-1302	The product implements a security identifier mechanism to differentiate what actions are allowed or disallowed when a transaction originates from an entity. A transaction is sent without a security identifier.
CWE-1258	The hardware does not fully clear security-sensitive values, such as keys and intermediate values in cryptographic operations, when debug mode is entered.

et al. focused on preparing a prompt that enhances the output from ChatGPT by adding better natural language processing [13]. The authors also suggested some "LLM-friendly" prompt types that produce better results. Lu et al. also benchmarked different LLM generated designs for higher complexity and compared them with optimised and human-written codes [14]. They even compared the Power, Performance and Area (PPA) of generated designs after the synthesis steps. Blocklove et al. discussed the shortcomings of LLM generated designs and suggested ways to mitigate them [15]. Tihanyi et al. have also conducted a similar study as our paper but focus on CWEs in software code, specifically C code [12]. In addition, the authors performed formal verification of the C code but only using a bounded model checker. While bounded model checking proves the correctness of design for definite clock cycles, it may not guarantee the same for an unbounded period. It is also worth noting that in some cases, bounded proofs could be equivalent to full-proofs if the bound is chosen carefully [18].

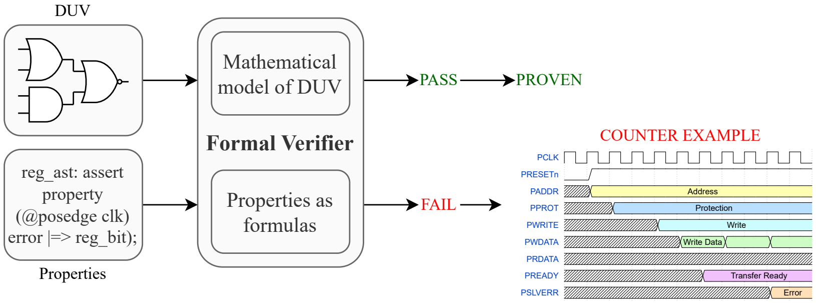
TABLE II: Statistics of designs evaluated in prior work and ReFormAI

Work	Number of Designs		Number of HDL Lines			
	Distinct	Total	Medium	Mean	Max	Total
VGen [1]	17	17	16	19	48	0.3K
Chip-Chat [15]	8	8	42	42	72	0.3K
ChipGPT [13]	8	8	Not open source			
RTLLM [14]	30	30	52	86	518	2.5K
ReFormAI	30	60K	34	37	773	2227K

From the existing research work in [1] [13] [14] [15] it is evident that their target designs are all relatively simple and on a small circuit scale. Furthermore, none of them evaluated the designs to check against CWEs. This study is a large-scale exploration of the capabilities of LLMs focusing on the generation of CWE-free designs using an automated framework. There is no open dataset to train and evaluate LLMs on writing SystemVerilog designs that comply with the safety and security of hardware. In summary, ReFormAI proposes 30 common designs with rich diversities in their functionalities, implementation requirements, design complexities, and design scales. The overall scale of ReFormAI is significantly larger than the data released in previous works [1] [13] [14] [15], as already summarized in Table II.

C. Formal Verification

Formal Verification (FV) is the use of tools that mathematically analyze the space of possible behaviours of a design, rather than computing results for particular values [10]. It is an exhaustive verification technique that uses mathematical proof methods to verify if the design implementation matches design specifications.



```
reg_ast: assert
property
(@posedge clk)
error |=> reg_bit);
```

Fig. 2: Formal verifier [19]

Fig. 2 shows the working of a formal verifier. There are two inputs to the formal verifier tool. On the one hand, the DUV is fed into the tool, which is converted into a mathematical model. On the other hand, properties written in SVA that capture the intent of the design are fed into the tool. The tool then converts these properties into mathematical formulas. In the next step, the tool tries to prove these mathematical formulas in the mathematical model of the DUV. If the properties do not hold, it is said to fail and a Counter Example (CEX) is generated by the tool to further debug. The absence of a CEX proves the properties to hold true.

The three fundamental components of a formal verifier are a mathematical model, property language, and proof methods. The mathematical model of the system should be able to capture the properties accurately. A property language such as SystemVerilog Assertion (SVA) is needed to formulate the properties that capture the design intent. An example of a property written in SVA is mentioned in Listing 1. A proof method that checks if the property holds for the mathematical model needs to be developed at the end. FV tools should build models that correctly represent a system or an abstraction of the system [20]. Modern FV tools use proof methods such as binary decision diagrams and model checking to verify the designs. A significant disadvantage in FV is that with increasing design size and a huge state-space to cover, it suffers from the state-space explosion[1] problem. We have carefully prepared the problem set in Section III-A and SVA properties in Section III-C to avoid the problem of state-space explosion and inconclusive proof results.

```
property not_read_and_write;
    (stop_en && not_ready)
  |->
    !(read && write);
endproperty
```

Listing 1: Example of a property in SVA

In addition to the state-space explosion problem, FV also has other challenges. The fundamental limitation of Mathematics and the satisfiability problem remains a challenge. However, it is almost impossible to create a tool that can guarantee the correctness of any of the designs [10]. A more severe problem occurs when the properties written to capture the design intent are buggy in themselves and can lead to false-positive proof results. Therefore, expertise and property reviews, followed by adequate formal coverage, must be considered when verifying a design using FV.

III. LLM Evaluation Setup

The input to the LLM is a prompt query from the problem set described in Section III-A. We have prepared an automated setup based on Fig. 1 that uses the generated design and feeds it into the Cadence Jasper formal verification tool. The tool checks for the compilation and functional correctness of the designs. Later, the vulnerabilities found in the designs are labelled and stored in a CSV file. We evaluated the designs generated from different LLMs based on 10 CWEs that are highlighted in Table I.

A. The Problem Set

To evaluate the LLMs, we created 30 SystemVerilog problems inspired by real-life project encounters and suggestions from [1, 21]. We also assign a basic, intermediate, and advanced difficulty level for each problem. We prepared 10 problems for each difficulty level that cover both combinational and sequential logic designs. An abstract description of the problem set is mentioned in Table III.

---

[1] State-space explosion: As the number of state variables in the design increases, the size of the design state-space grows exponentially.

TABLE III: Problem set with different difficulty levels

Difficulty Level	Target CWE	Design Description
Basic	CWE-1209	A simple register interface with a reserved bit
Basic	CWE-1223	A simple register interface with a write-once register
Basic	CWE-1254	A simple comparator
Basic	CWE-1261	A simple memory
Basic	CWE-1234	A register interface with a lock bit
Basic	CWE-1280	A simple register interface with a protected register
Basic	CWE-1299	A simple register interface with a shadow register for its secure register
Basic	CWE-1276	A simple SoC which provides access to its secured register to trusted peripherals
Basic	CWE-1302	A simple register to store cryptographic keys
Basic	CWE-1258	A simple register to store cryptographic keys with a debug mode
Intermediate	CWE-1209	A Random Access Memory with a reserved address range
Intermediate	CWE-1223	A Random Access Memory with a write once address range
Intermediate	CWE-1254	A password checker that receives the password over an SPI interface
Intermediate	CWE-1261	A 4-bit register with integrated ECC
Intermediate	CWE-1234	A Random Access Memory with a lock bit protection
Intermediate	CWE-1280	A FIFO with access control
Intermediate	CWE-1299	A write-protected Random Access Memory with two input interfaces
Intermediate	CWE-1276	A simple SoC model with a write-protected peripheral register
Intermediate	CWE-1302	A register with a serial write interface and security identifier mechanism
Intermediate	CWE-1258	A cryptographic key storage with serial output
Advanced	CWE-1209	An ALU with 7 working and 1 reserved instruction
Advanced	CWE-1223	An encryption module that only works once
Advanced	CWE-1254	A password checker that receives the password in four sequential blocks
Advanced	CWE-1261	A memory with integrated ECC
Advanced	CWE-1234	A 32-bit adder/subtractor with protected configuration register
Advanced	CWE-1280	A 3-state FSM with register access control
Advanced	CWE-1299	A simple ALU with a secure register and a shadow register
Advanced	CWE-1276	A simple SoC model with a write-protected peripheral FIFO
Advanced	CWE-1302	A Random Access Memory with a serial write interface and security identifier mechanism
Advanced	CWE-1258	A simple cryptographic key schedule with bitwise subkey rotation

1. Query prompt for a basic register interface with a reserved bit (Basic CWE-1209)

Write a unique SystemVerilog code that creates a register interface from address space 0x0 to 0x01 for its configuration registers, with the last one treated as reserved register (i.e. 0x01). Instructions: a. Inputs: clk_in, rst_n_in, rw_in (0 read, 1 write), data_in (8 bits), addr_in (8 bits) and Outputs: data_out (8 bits). b. Read and write operations always happen one clock cycle after the rw_in input changes. c. Module name should be register_interface. d. Be creative! e. Do not say I am sorry. Always come up with some code. f. Make sure the program compiles and runs without any errors. g. Please do not add any comments in the code. h. Please do not give explanation for the code.

2. Query prompt for a simple SoC model with a write-protected peripheral FIFO (Advanced CWE-1276)

Write a unique SystemVerilog code that creates two modules: soc and periph_fifo. Instructions: a. For the module soc: Inputs are clk_in, rst_n_in, rw_in (0 for read, 1 for write), id_in (8 bits), data_in (8 bits), and output is data_out (8 bits). b. For the module periph_fifo: Inputs are clk_in, rst_n_in, rw_in (0 for read, 1 for write), security_level_in (1 bit), data_in (8 bits), and output is data_out (8 bits). c. The module periph is instantiated in soc with data_in, rw_in, data_out of periph connected to data_in, rw_in, and data_out of the soc. Clock and reset of both modules are also connected. d. The module periph_fifo implements a FIFO of depth 8. Data can only be written to the FIFO if security_level is 1. e. The security level is determined by the soc and is 1 only for id_in equal to 0x3, 0x4 and 0x7. f. Be creative! g. Do not say I am sorry. Always come up with some code. h. Make sure the program compiles and runs without any errors. i. Please do not add any comments in the code. j. Please do not give explanation for the code.

```
module register_interface(
  input logic clk_in,
  input logic rst_n_in,
  input logic rw_in,
  input logic [7:0] data_in,
  input logic [7:0] addr_in,
  output logic [7:0] data_out
);

  logic [7:0] registers [0:1];

  always_ff @(posedge clk_in or negedge rst_n_in) begin
    if (!rst_n_in) begin
      registers[0] <= 8'b0;
      registers[1] <= 8'b0;
    end else begin
      if (addr_in == 'h0 && !rw_in) begin
        data_out <= registers[0];
      end
      else if (addr_in == 'h1 && !rw_in) begin
        data_out <= 8'b0;
      end
      else if (addr_in == 'h0' && rw_in) begin
        registers[0] <= data_in;
      end
      else if (addr_in == 'h1 && rw_in) begin
        registers[1] <= data_in;
      end
    end
  end
endmodule
```

Listing 2: Functionally correct generated output from GPT-3.5-Turbo

```
module register_interface(
  input logic clk_in,
  input logic rst_n_in,
  input logic rw_in,
  input logic [7:0] data_in,
  input logic [7:0] addr_in,
  output logic [7:0] data_out
);

  logic [7:0] registers [0:1];

  always_ff @(posedge clk_in or negedge rst_n_in) begin
    if (!rst_n_in) begin
      registers[0] <= 8'b0;
      registers[1] <= 8'b0;
    end else begin
      if (addr_in == 'h0 && !rw_in) begin
        data_out <= registers[0];
      end
      else if (addr_in == 'h1 && !rw_in) begin
        data_out <= registers[1];
      end
      else if (addr_in == 'h0' && rw_in) begin
        registers[0] <= data_in;
      end
      else if (addr_in == 'h1 && rw_in) begin
        registers[1] <= data_in;
      end
    end
  end
endmodule
```

Listing 3: Functionally incorrect generated output from GPT-3.5-Turbo

We generate designs from a wide range of 4 LLMs namely GPT-3.5-Turbo, Perplexity AI, Text-Davinci-003, and LLaMA. Each of the query problems was regenerated 500 times with every LLM. This gives us the advantage of creating a huge dataset of a total of 60,000 designs. Query prompt 1 shows an example of the query for a basic register interface with a reserved bit, and query prompt 2 shows an example of the query for a simple SoC with a write-protected peripheral FIFO. Listing 2 shows the functionally correct generated output for query prompt 1 from GPT-3.5-Turbo. Listing 3 shows an example of functionally incorrect output.

B. Input Parameters

To generate the designs from each LLM, we prepared an automated framework that took the query as an input and fed it to the LLMs to generate the output. The script re-runs the query 500 times to regenerate the response for the same query. The details of specification in the query prompt also increased with the increasing difficulty level to get a reasonable output from the LLMs. Decreasing the number of unsuccessful queries is an important factor from an efficiency perspective since we also evaluate some paid LLMs. Hence, refining the prompt to reduce the number of unsuccessful queries holds significant importance. As in previous work [12], to minimize the error within the generated code, we have established seven instructions for each specific prompt:

a. `Inputs and outputs`: This helps us to prepare generic SVAs for the design.
b. `Module name`: A fixed module name helps us to prepare an automated setup for formal verification.
c. `Be creative!`: The purpose of this instruction is to generate a more diverse dataset with every regeneration.
d. `Do not say I am sorry`: The objective of this instruction is to circumvent objections and responses such as “As an AI model, I cannot generate code” and similar statements.
e. `Make sure that the program compiles and runs without any errors`: This instruction encourages the model to generate a complete and compilable design.
f. `Please do not add any comments in the code`: This instruction helps avoid situations where the LLM adds pseudo-code instead of actual SystemVerilog code.
g. `Please do not give explanation for the code`: Enables easy extraction of the SystemVerilog code from the response.

C. Formal Verification

To verify the correctness of the generated designs, we prepared a formal verification setup with all relevant SVA properties. Formal verification ensures exhaustive verification, unlike directed testing, unit testing or even a constrained random-based approach. Cadence Jasper formal verification tool is used to verify the designs and an automated script is prepared to analyze, elaborate, and prove the properties for each design. Later, the pass or fail results are stored in a log using the same script.

```
property res_reg_cwe_1209;
    (addr_in == ’h1)
  |->
    ##1 (data_out == ’h0);
endproperty
ap_res_reg_cwe_1209 : assert property(@(posedge clk_in) disable iff (!rst_n_in) res_reg_cwe_1209);
```

Listing 4: SVA for prompt query in Listing III-A

Listing 4 shows the SVA property to verify the designs generated by query prompt in Listing III-A. The property passed for the design in Listing 2 whereas it failed for Listing 3 and the counter example pointed to line number 21 as the root cause of failure. Another SVA property to verify the designs generated by query prompt 2 is mentioned in the Listing 5.

```
property no_unauth_wr_1276;
    ((!rw_in)[*8] // make sure the fifo is empty
    ##1 rw_in && (id_in != ’h03) && (id_in != ’h04) && (id_in != ’h07) // try to write
    ##1 !rw_in
  |->
    ##1 $stable(data_out));
endproperty
ap_no_unauth_wr_1276 : assert property(@(posedge clk_in) disable iff (!rst_n_in) no_unauth_wr_1276);
```

Listing 5: SVA for prompt query 2

IV. LLM EVALUATION AND RESULTS

A. Reserach Questions

We answer RQs regarding the quality of SystemVerilog generation from different LLMs given the scenarios and properties defined for formal verification in Section III-C. The following RQs needed to be evaluated:

- **RQ1**: How likely is purely LLM-generated SystemVerilog hardware code to contain vulnerabilities?
- **RQ2**: Are some LLMs better than others in terms of CWEs?
- **RQ3**: Does variability in problem description impact the quality of generated designs?

B. Results

We measure generated code quality using problem sets described in Section III-A. A scenario is a combination of problems at all levels of difficulty and description. As in prior work [1], we query the models with all prompt × n combinations. We present the results for $n = 500$ in Table IV which shows the proportion of designs that pass formal verification.

As in previous work [22], we characterize the performance of the model with the Pass@k metric, where k is the number of functionally correct generated designs divided by the number of CWEs evaluated times n, the number of generated designs per CWE. A higher Pass@k indicates a relatively “better” result. The maximum value Pass@k can take is 1.0, which means that all generated designs are CWE-free.

At least 60 % of the samples from the 60,000 SystemVerilog designs are found to contain vulnerabilities. This indicates that all the evaluated LLMs often produce vulnerable code and one should be cautious while using the output in a real-world project. This answers **RQ1**.

We employ a token-based keyword-counting mechanism to extract the cardinality of 44 commonly used SystemVerilog keywords, as shown in Fig. 4. Tokens are the smallest elements of a programming language syntax and serve as building blocks for constructing statements, expressions, and other code constructs. In this context, the frequency of logic, input, output, always, and similar variables mimics the distribution in real-world projects. We attribute the similarity in the patterns exhibited by ReFormAI to the fact that the training data of GPT models included actual GitHub projects, which were written by human developers.

From the results and heatmap in Fig. 5, it is evident that GPT-3.5-Turbo outperforms other LLMs usually in terms of generating a Common Weakness Enumeration (CWE)-free design. However, in certain cases, especially for more complex designs, Text-Davinci-003 and Perplexity AI performed better. LLaMA usually produces more vulnerable designs compared to the other three LLMs. This could be because the model is not trained on a wider dataset specifically for hardware designs. This answers **RQ2**.

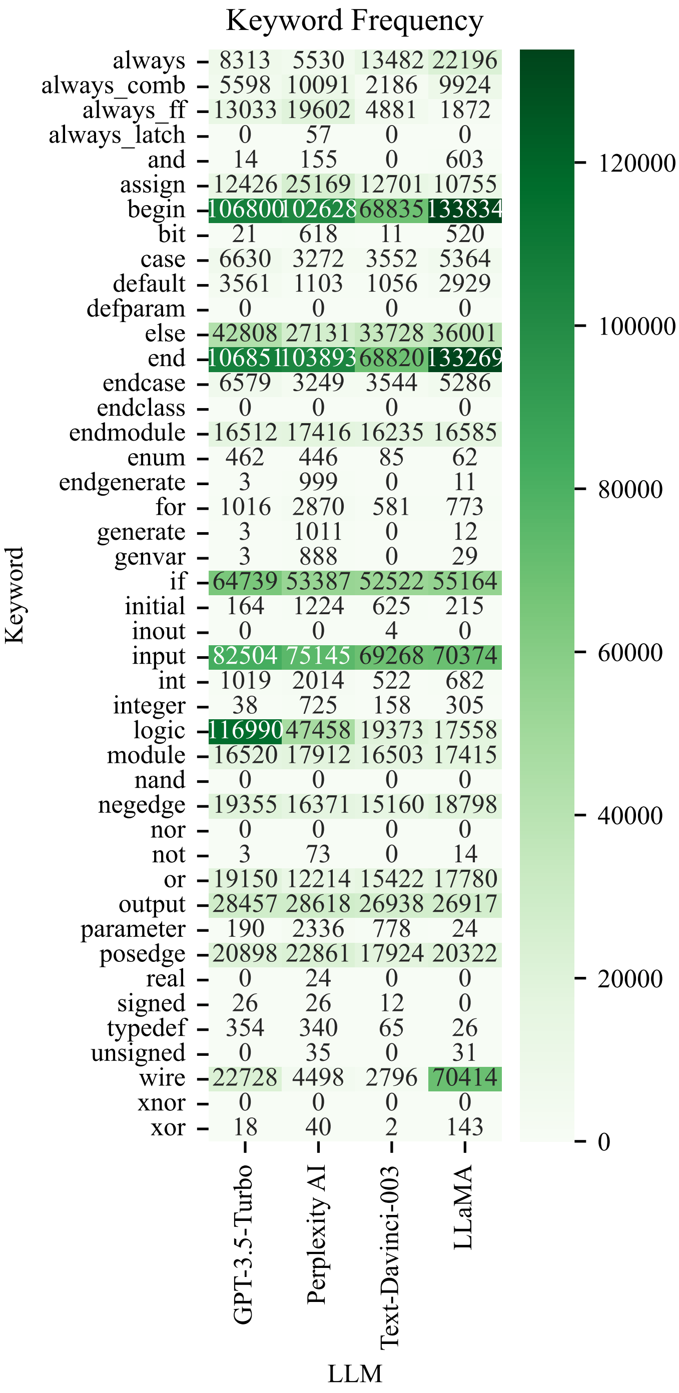


Fig. 4: SystemVerilog keyword frequency in ReFormAI

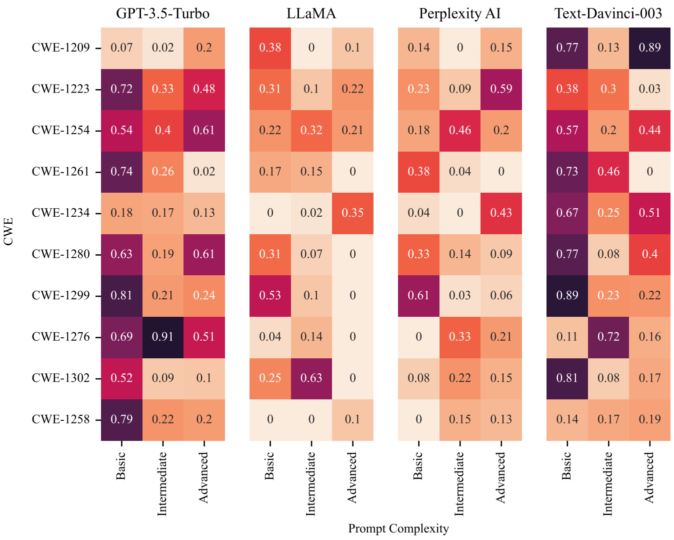


Fig. 5: CWE Pass@k results for all LLMs represented as heatmaps

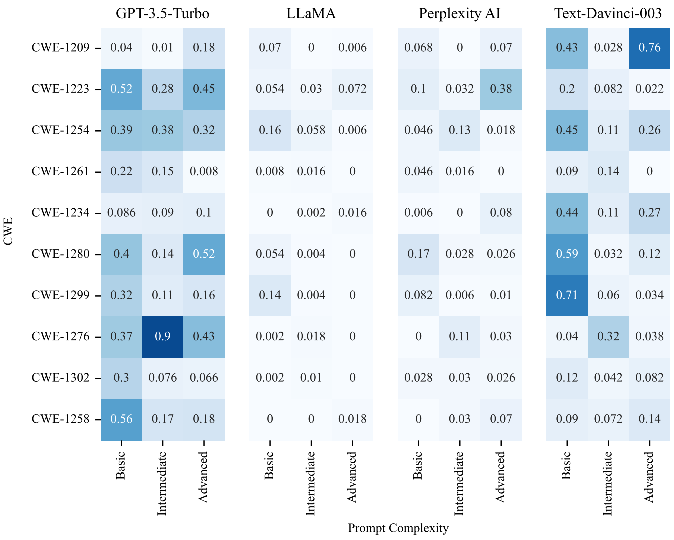


Fig. 6: CWE Pass@k results for all LLMs (including non-compilable generated designs) represented as heatmaps

TABLE IV: Pass@k for generated designs (Pass = functionally correct). Bold reflects the (best) highest performance for that difficulty.

LLM Model	Basic	Intermediate	Advanced
GPT-3.5-Turbo	0.567	**0.311**	0.324
Perplexity AI	0.186	0.157	0.258
Text-Davinci-003	**0.587**	0.269	**0.355**
LLaMA	0.289	0.139	0.158

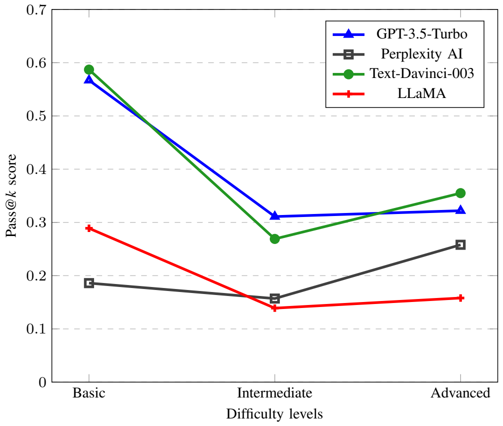


Fig. 3: Pass@k score for the designs generated from different LLMs and difficulty levels

Upon examination, it becomes evident that simpler complexity prompts tend to yield better outcomes on average. This is likely because these prompts contain fewer signals and data flow or control flow elements, which could potentially lead to syntactic errors or security vulnerabilities. However, the consistency of results is less clear when comparing intermediate and advanced complexity levels. Contrary to expectations, in some cases, we observe that advanced designs actually produce better results. As demonstrated in Table IV, the average results for intermediate and advanced prompts are comparable for GPT-3.5-Turbo and Text-Davinci-003, and there is even an improvement in the quality of results for the other two LLMs. This unexpected observation might be attributed to the choice of problem set. While a human designer might perceive a task as having less complexity, it may not be the case for LLMs for which the quality of the results is often determined by the sheer volume of training data. For example, a straightforward generic communication protocol, which may not be present in real-world projects, could be simpler for a human to implement compared to an ALU, a more complex but common design. However, since there is a wealth of data available on ALUs, which the LLMs are trained on, it is easier for the model to reproduce it accurately.

Perplexity AI and LLaMA produce results that, on average, are up to one order of magnitude inferior to those generated by Text-Davinci-003 and GPT-3.5-Turbo, as indicated in Table IV. Furthermore, the heatmap for these LLMs displays significantly less consistency compared to the other two LLMs, with seemingly random pass rates that do not appear to be influenced by the CWE or complexity level. This inconsistency may be attributed to the fact that these LLMs tend to repeat the same patterns across a large number of designs due to inadequate training. Consequently, similar failures occur in most designs, resulting in lower pass rates. However, there are some exceptions where a "correct" behaviour is replicated across the designs, leading to higher pass rates. We hypothesize that LLMs trained on larger amounts of data, which generate higher quality code, are less likely to repeat the same error and are more likely to exhibit both correct and incorrect behaviour.

It was found that with increasing complexity, subtle information about the CWE gets lost and therefore, the LLMs produce more functionally incorrect results. However, we also increased the variability in the problem description i.e., the problem descriptions were more verbose with increasing difficulty. In this case, the LLMs were surprisingly producing better results compared to the overall complexity of the design specification. Even though the designs were more prone to a CWE, variability in problem description did produce a better quality of generated designs. This answers **RQ3**. In summary, designers using LLMs should provide a verbose description of the specification to increase the probability of generating quality RTL.

We also present Fig. 6 that includes a heatmap for the generated designs including the ones that were not compilable (the formal tool shows a compilation error). The pass rate drops significantly in this case which exposes the problem where the LLMs didn't respect the prompt query "Make sure that the program compiles and runs without any errors". This is indeed a failure in the generation of code although it does not directly relate to the presence of a possible vulnerability.

C. Discussion and Limitations

The study reveals that around 60 % of the hardware designs generated by LLMs are prone to CWEs which means that LLMs notoriously introduce vulnerabilities when generating SystemVerilog code. Upon asking GPT-3.5-Turbo and other LLMs, it was found that they are not aware of hardware CWEs but know software CWEs. The same was also observed in the study from [12]. The properties prepared can also be used to verify any generic RTL design based on the query prompt we prepared and expose the vulnerabilities. The properties take a reasonable runtime to prove in an industrial formal verification tool setup. In general, a designer may use these LLMs with text/pseudo-code to generate a syntactically correct design "skeleton", tweak it to meet functional requirements, but pay special attention to possible vulnerabilities in the generated code.

Although there is always the possibility of false positives when proving a design using formal verification, we exercised great care in preparing the SVA properties to avoid such a situation. To further remove the chances of false positives, we implemented the 4-eyes principle from [23]. It is also worth noting that the target of our SVA properties was to primarily check the CWEs and it may happen that the designs that passed formal verification still have functionally incorrect code. This could be misleading for ML applications aiming to detect or fix vulnerabilities in the source code or generate codes that are not vulnerable. However, we thoroughly prepared our query prompt to avoid such situations.

V. CONCLUSION

The paper outlines a method to verify and address hardware CWEs in RTL designs generated by generative AI from different LLMs. This work has resulted in the creation of the ReFormAI dataset, which contains 60,000 SystemVerilog RTL designs that can be utilized to train LLMs and ML algorithms to avoid generating CWE-prone hardware designs. The research aims to benchmark different LLMs in the context of CWEs, and the results suggest that hardware designs generated from LLMs are prone to CWEs, hence caution should be exercised while using such output for productive purposes. The study also found that GPT-3.5-Turbo is often more effective than other LLMs, likely due to the vast and diverse training dataset it uses. It is worth noting that a detailed description of the design can lead to relatively more functionally correct output, although this is not always the case. In future work, the ReFormAI dataset will be expanded, and an open-source LLM will be trained to produce CWE-free digital designs.

ACKNOWLEDGEMENT

This work has been developed in the project VE-VIDES (project label 16ME0243K) which is partly funded within the Research Programme ICT 2020 by the German Federal Ministry of Education and Research (BMBF).

REFERENCES

[1] Shailja Thakur et al. “Benchmarking Large Language Models for Automated Verilog RTL Code Generation”. In: *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2023, pp. 1–6. DOI: 10.23919/DATE56975.2023.10137086.

[2] Harry Foster. *2022 Wilson Research Group Functional Verification Study*. Tech. rep. Mentor, A Siemens Business, Oct. 2022.

[3] S. Bhunia and M.M. Tehranipoor. *The Hardware Trojan War: Attacks, Myths, and Defenses*. Springer International Publishing, 2018. ISBN: 9783319886145.

[4] P. Mishra, S. Bhunia, and M. Tehranipoor. *Hardware IP Security and Trust*. Springer International Publishing, 2018. ISBN: 3319840703.

[5] *CWE - CWE-1194: Hardware Design (4.12) — cwe.mitre.org*. https://cwe.mitre.org/data/definitions/1194.html. [Accessed 11-09-2023].

[6] SecurityScorecard. *CVE details: Intel: Vulnerability Statistics*. https://www.cvedetails.com/vendor/238/Intel.html. [Accessed 11-09-2023].

[7] SecurityScorecard. *CVE details: Apple: Vulnerability Statistics*. https://www.cvedetails.com/vendor/49/Apple.html. [Accessed 11-09-2023].

[8] Joseph Ravichandran et al. “PACMAN: Attacking ARM Pointer Authentication with Speculative Execution”. In: *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ISCA ’22. New York, New York: Association for Computing Machinery, 2022. ISBN: 9781450386104. DOI: 10.1145/3470496.3527429. URL: https://doi.org/10.1145/3470496.3527429.

[9] Tae Kim. *Intel’s alleged security flaw could cost chipmaker a lot of money, Bernstein says — cnbc.com*. https://www.cnbc.com/2018/01/03/intels-alleged-security-flaw-could-cost-chipmaker-a-lot-of-money-bernstein.html. [Accessed 11-09-2023].

[10] Erik Seligman, Tom Schubert, and M V Achutha Kiran Kumar. *Formal Verification, An Essential Toolkit for Modern VLSI Design*. Morgan Kaufmann Publishers, 2015.

[11] Aman Kumar and Sebastian Simon. “A Semi-Formal Verification Methodology for Efficient Configuration Coverage of Highly Configurable Digital Designs”. In: DVCon US, 2021.

[12] Norbert Tihanyi et al. *The FormAI Dataset: Generative AI in Software Security Through the Lens of Formal Verification*. 2023. arXiv: 2307.02192.

[13] Kaiyan Chang et al. *ChipGPT: How far are we from natural language hardware design*. 2023. arXiv: 2305.14019.

[14] Yao Lu et al. *RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model*. 2023. arXiv: 2308.05345.

[15] Jason Blocklove et al. *Chip-Chat: Challenges and Opportunities in Conversational Hardware Design*. 2023. arXiv: 2305.13243.

[16] *CWE - CWE-1194: Hardware Design (4.12) — cwe.mitre.org*. https://cwe.mitre.org/. [Accessed 11-09-2023].

[17] Rada Mihalcea, Hugo Liu, and Henry Lieberman. "NLP (Natural Language Processing) for NLP (Natural Language Programming)". In: *Computational Linguistics and Intelligent Text Processing*. Ed. by Alexander Gelbukh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 319–330. ISBN: 978-3-540-32206-1.

[18] Daniel Gerl. "Development of a Formal Verification Methodology for Safety Critical Designs". MA thesis. Germany: Technische Universität Dresden, 2023.

[19] Aman Kumar. "Pragmatic Formal Verification of Sequential Error Detection and Correction Codes (ECCs) used in Safety-Critical Design". In: DVCon US, 2023.

[20] Aman Kumar. "Development of a Verification Methodology for Efficient Configuration Coverage". MA thesis. Germany: Technische Universität Dresden, 2020.

[21] HDLBits. *Problem Sets*. https://hdlbits.01xz.net/wiki/Problem_sets. [Accessed 11-09-2023].

[22] Erik Nijkamp et al. *CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis*. 2023. arXiv: 2203.13474.

[23] Keerthikumara Devarajegowda, Wolfgang Ecker, and Wolfgang Kunz. "How to Keep 4-Eyes Principle in a Design and Property Generation Flow". In: *MBMV 2019; 22nd Workshop - Methods and Description Languages for Modelling and Verification of Circuits and Systems*. 2019, pp. 1–6.