LLM-assisted Generation of Hardware Assertions

Rahul Kande*, Hammond Pearce†, Benjamin Tan‡, Brendan Dolan-Gavitt§,
Shailja Thakur§, Ramesh Karri§, and Jeyavijayan Rajendran*
*Texas A&M University, †University of New South Wales, ‡University of Calgary, §New York University
*{rahulkande,jv.rajendran}@tamu.edu, †hammond.pearce@unsw.edu.au,
‡benjamin.tan1@ucalgary.ca, §{brendandg,st4920,rkarri}@nyu.edu

***Abstract*—The security of computer systems typically relies on a hardware root of trust. As vulnerabilities in hardware can have severe implications on a system, there is a need for techniques to support security verification activities. Assertion-based verification is a popular verification technique that involves capturing design intent in a set of assertions that can be used in formal verification or testing-based checking. However, writing security-centric assertions is a challenging task. In this work, we investigate the use of emerging large language models (LLMs) for code generation in hardware assertion generation for security, where primarily natural language prompts, such as those one would see as code comments in assertion files, are used to produce SystemVerilog assertions. We focus our attention on a popular LLM and characterize its ability to write assertions out of the box, given varying levels of detail in the prompt. We design an evaluation framework that generates a variety of prompts, and we create a benchmark suite comprising real-world hardware designs and corresponding golden reference assertions that we want to generate with the LLM.**

***Index Terms*—LLM, AI, hardware, assertion generation, assertions, hardware security, vulnerability detection**

I. Introduction

A. Implications of Vulnerable Hardware

Hardware underpins applications ranging from small internet-of-thing (IoT) devices to large and complex multi-core processors. Many of the tasks traditionally performed in software, like encryption, decryption and machine learning, are accelerated in hardware. Many software security defenses are built assuming hardware as the vulnerability-free root of trust. Vulnerabilities in hardware have severe implications on all such applications and security defences [1]–[4], yet weaknesses continue to arise (e.g. Spectre [5], Meltdown [6], RowHammer [7]).

The cost of fixing hardware vulnerabilities depends on where in the development lifecycle they are discovered. Vulnerabilities discovered after fabrication cannot usually be fixed. Even if software can patch them, they incur performance overheads. Vulnerabilities found in the field have consequences ranging from information leakage to corporate reputation damage. Hence, it is desirable to detect as many vulnerabilities as possible during hardware design.

B. State-Of-The-Art Hardware Verification

State-Of-The-Art Hardware Validation and Verification techniques include testing [8] (simulation, random-regression, directed-random testing) and formal verification [9], [10] (model-checking, theorem proving). To meet the demand for faster and efficient verification, researchers developed techniques like hardware fuzzing [11], [12], information-flow tracking [13], and hybrid techniques [14] that combine approaches. These techniques require a golden reference model (GRM) or hardware assertions to detect vulnerabilities. Generating GRMs or assertions is not trivial requiring manual effort and knowledge of the design. Such methods are error-prone and do not scale.

C. Assertion-Checking

Assertion-checking [15] is a popular verification technique where the specification of the design under test (DUT) is coded into assertions or properties in a hardware description language (HDL) like SystemVerilog. These assertions are used to statically prove properties using formal verification tools or dynamically verify using testing tools. Assertions play an important role in hardware verification. Each assertion will focus on verifying individual functional properties and critical logic in the hardware. They can detect vulnerabilities in the early stages of design, even when the DUT is not fully developed, as they do not require the output of the DUT to detect vulnerabilities. They can also detect vulnerabilities that will occur when components are composed [16]. Assertions can be customized to target the goal of verification. This is unlike code coverage which focuses on functional verification of the DUT. Such assertions can guide hardware fuzzers and accelerate verification.

D. Limitations of Assertion-Checking

Generating correct and practical assertions is a challenging task. Automatic derivation (i.e. mining) for assertions [17], [18]) is possible, irrelevant and difficult checks will increases verification time. While assertions for functional behaviors check that the DUT performs to expectations, security assertions typically check that a DUT *does not* feature some weakness, a fundamentally different task and one not well-suited for mining [19]. Ensuring that assertions check for security vulnerabilities requires knowledge of the DUT and analysis of the potential weaknesses it may have. This requires security expertise unavailable to a typical designer. Thus, techniques to develop security assertions are error-prone time-consuming and do not scale to large designs. Even with security-relevant knowledge, say in natural language descriptions of properties, this knowledge is not used to automatically write assertions.

E. Goals and Contributions

To encourage adoption of assertion-based security checking, it is essential to determine faster and easier methods of generating hardware security assertions. In this paper, we investigate the use of *Large Language Models (LLM)* to generate hardware security assertions automatically. Given their relative success in writing code for other languages (e.g., OpenAI's Codex [20]), we examine if LLMs can be utilized for this task. This is the first work investigating the feasibility of LLMs to generate hardware security assertions. We envisage a pipeline where designers/verification engineers write comments in natural language about the security assertions, for example, based on system specifications, in the RTL code. These comments are used with surrounding context as a *prompt* by the LLMs to generate the security assertions.

We study commercial LLMs and answer two research questions: **(RQ1)** Is it possible for LLMs to generate security assertions for hardware? **(RQ2)** How do LLMs perform with respect to different types of prompts? We developed a framework to evaluate LLM performance when tasked with writing security assertions, including designing a benchmark suite of different types of vulnerabilities in hardware. Our main contributions are fourfold:

1.评估LLM生成的硬件断言，可以修复一些断言问题
2.评估LLM的性能
3.调查不同prompt对于断言生成的影响

- A framework and a set of benchmarks to evaluate LLMs in generating hardware assertions. The framework can fix a limited set of syntax/typographical issues in the assertions,
- Evaluating a popular LLM for generating security assertions
- Investigating the effect of different types of prompts on the generation of assertions, and
- Open-sourcing LLM-based framework and benchmarks [following review] to support research in this direction.

II. Background and Related Work

A. Security Assertions

Assertion-based verification is a popular part of the digital design flow, where designer intent is captured as a set of properties that are checked during simulation, through formal verification, or synthesized into actual hardware for run-time checking. Assertions help ascertain whether a given property is satisfied, and this guides designers to vulnerabilities that require fixes. Designers often use assertions to check functional requirements [15]; this is a non-trivial task that requires expertise and considerable manual effort in the design and verification of hardware. Writing security assertions to catch *security* vulnerabilities provides additional challenges [21].

B. Large Language Models for Code Generation

The domain of "natural language programming" [22] seeks to transform natural language specifications into code through natural language processing. Large Language Models (LLMs) such as GPT-2 [23] and BERT [24], are transformer-based artificial neural networks [25] which are designed to work over text datasets. LLMs have millions to billions of parameters and are trained over expansive datasets of text. Inputs and outputs for an LLM are in the form of tokens, themselves common sets of character sequences. Each token has a unique numeric identifier (also known as a byte pair encoding [26]). Functionally, given a sequence of tokens as an *input prompt*, an LLM will output a probability distribution for the next token over the vocabulary of known tokens. After a token is selected based on some search criteria, it is appended to the prompt. The LLM then generates the next token. This is known as auto-regression. The sequence of tokens generated from the *input prompt* is known as the "output" or "completion." Users can optionally specify a "stop sequence" – a sequence of tokens that tells the model (API) to stop generating tokens.

Since LLMs were trained on regular text, some LLMs have been trained to operate over programming language codes. OpenAI Codex LLM [20] elucidates functional code from program snippets such as comments and function signatures. OpenAI Codex and GitHub Copilot [27] are commercial tools. Codex was trained over "all" the open-source code on GitHub, i.e. millions of code files with hundreds of millions to billions of lines of code. As such, this LLM "learned" to support languages with an open-source presence, including Verilog [28], [29]. Other open-source LLMs include NVIDIA MegatronLM [30] and Salesforce CodeGen [31]. Although training an LLM from scratch is expensive, finetuning an LLM is accessible, and several LLMs finetuned on individual languages have been open-sourced. Pearce et al. fine-tuned a GPT-2 model to produce "DAVE" [32], a "small" LLM that produces Verilog from a small set of natural language descriptions.

C. Automating Assertion Generation

Aside from generating new designs, related work explores specialized parts of the design flow, such as patch generation (program repair) [33], [34]. Our work similarly focuses on the security assertion generation problem in the hardware design flow. Given the aforementioned challenges in designing and verifying hardware designs, prior work has investigated machine learning and other automated techniques to assist designers. For example, GoldMine [17] mines invariant properties from static analysis and analysis from simulation traces. Recent work has used similar approaches to focus on security-related properties (e.g., [19], [35], [36]). Readers can refer to a recent survey from Witharana et al. [15] for other examples.

Other prior work in automated assertion generation investigated the mapping between natural language specification and SystemVerilog Assertions on a small scale [37] and focused on learning a custom grammar (representing a "writing style") for use in a translation system that maps sentences to assertions. Harris and Harris' work was not specific to security. In contrast, Zhang and Sturton's Transys [38] focuses on security properties. In that work, security properties from one design are translated to another in an automated fashion in a series of passes that identify variables in the first design and their analogs in the other, adjust the arithmetic expressions in the property, and then refine the constraints for the property. Tran-

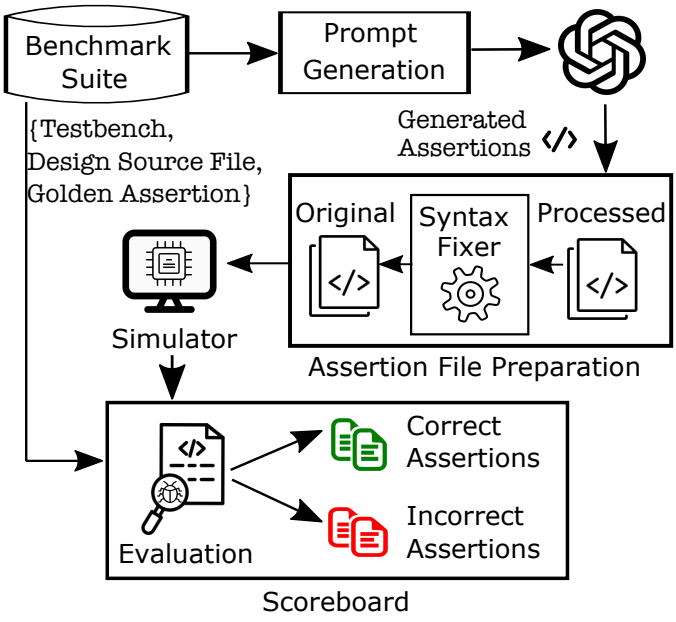

Figure 1. Evaluation framework for our assertion generator.

sys requires properties to start from and makes its mapping of registers, signals, and ports between designs using a mix of statistic, semantic, and structural features.

Unlike the aforementioned works, we focus on mapping high-level designer intent (captured in natural language) to SystemVerilog assertions, without training, simulation, or assertions from closely-related projects, as part of the generation. Specifically, we evaluate the usefulness of an out-of-the-box LLM for this task.

III. Evaluation Framework: Assessing LLM For Assertion Generation

To measure the ability of an LLM to generate hardware security assertions, we design an evaluation framework, depicted in that first generates a prompt based on one of our benchmarks, then creates an assertion for a desired property in that benchmark by querying the LLM, and finally takes that generated output and simulates it for comparison against a "golden" reference assertion.

The evaluation framework has the following components: (i) benchmark suite, (ii) prompt generator, (iii) assertion file generator, (iv) simulator, and (v) scoreboard, as shown in Figure 1. Next, we explain each of these components in detail.

A. Benchmark Suite

The benchmark suite consists of two manually crafted designs and eight modules derived from the designs of Hack@DAC hardware security competitions [21], [39], [40] and the open source silicon root of trust SoC, OpenTitan [41] as shown in Table I. The table includes the target assertion we aim to generate and the vulnerability that is detected with these assertions. The common weakness enumerations (CWEs) corresponding to these vulnerabilities span several hardware

Listing 1. `goldenAssert.sva` file of BM1 showing its golden reference assertion (reformatted for length).

```
`timescale 1ns/10ps
module v_dut (
    data_in , data_out, r_en    , w_en    ,
    lock    , clk     , rst     , data
);
input data_in ;    input data_out;
input r_en    ;    input w_en    ;
input lock    ;    input clk     ;
input rst     ;    input data    ;

// golden assertion
assert property ( @(posedge clk) (data ^ $past(data))
↪  |-> ($past(lock) == 0) )
    else $display("GOLDEN: FAIL, time=%4d, data=%d,
    ↪  data_d=%d, lock=%d", $time, data, $past(data),
    ↪  $past(lock));

endmodule

bind tb
    v_dut
    i_bind_dut (
      .data_in  (data_in),
      .data_out (data_out),
      .r_en     (r_en),
      .w_en     (w_en),
      .lock     (lock),
      .clk      (clk),
      .rst      (rst),
      .data     (data)
  );
```

CWE categories such as (i) debug and test problems (CWE-1207), (ii) peripherals, on-chip fabric, and interface/IO problems (CWE-1203), (iii) privilege separation and access control issues (CWE-1198), and (iv) power, clock, thermal, and reset concerns (CWE-1206). Each benchmark includes the following information.

Design source code. Each benchmark includes the source code of the corresponding benchmark design. To limit the number of tokens in the prompt to LLM, the original source files from Hack@DAC and OpenTitan are trimmed down and simplified to less than 100 lines of code, retaining only the logic corresponding to the target vulnerability. The source code provides the relevant context about the target assertion to the LLM. Each benchmark includes three types of *design source code* as we describe below.

- **EmptySource** is an empty source file with no information about the benchmark design. An empty source code evaluates the ability of the LLM to generate the assertions in the absence of design context.
- **CorrectSource** is the correct source code of the benchmark. Correct source codes provides complete context about the design to the LLM.
- **BuggySource** is a source code with vulnerability in it. Benchmarks derived from Hack@DAC designs have vulnerabilities inserted in them while the vulnerabilities are manually added in all other benchmarks.

Golden reference assertion. Each benchmark consists of a reference assertion in a SystemVerilog assertion (SVA) file, `goldenAs- sert.sva`. These golden assertions are manually crafted for BM1-BM8, as there are no existing

Table I
BENCHMARKS USED FOR EVALUATION ALONG WITH THE ASSERTION TO GENERATE USING THE LLMS. THE VULNERABILITIES IN THE BENCHMARKS AND THEIR CORRESPONDING CWE CLASSIFICATIONS ARE ALSO INCLUDED.

ID	Benchmark	Source	Assertion	Vulnerability	CWE Type
BM1	Lockable Register	Manually crafted	Data should not change if the lock is set.	The register can be written even if it is locked.	CWE-1233
BM2	Traffic signal controller	Manually crafted	Yellow signal should precede RED signal.	The traffic controller skips the yellow light when pedestrians request to cross the road.	CWE-1245
BM3	JTAG password controller	Hack@DAC	Valid signal should not be asserted if JTAG is locked.	Able to access locked JTAG.	CWE-1324
BM4	Bus access control	Hack@DAC	Access values to each peripheral should match the specification.	Access to one peripheral grants access to another peripheral.	CWE-1317
BM5	AES IP	Hack@DAC	Data should not be output if the encryption is not completed.	Internal registers of AES are visible externally.	CWE-1303
BM6	AES IP	Hack@DAC	Key values should be cleared if we are entering debug mode.	Secret keys are not cleared when entering debug mode.	CWE-1244
BM7	Register controller of a CVA6 processor	Hack@DAC	Lower than required privilege level should trigger violation.	Privileged CSR register can be accessed by unprivileged user.	CWE-1261
BM8	Register lock IP	Hack@DAC	All the register locks should be initialized correctly on reset.	Register locks are configured with incorrect default values at reset.	CWE-276
BM9	ADC controller	OpenTitan SoC	The value of wakeup timer should be 0 on reset.	Wakeup timer is incorrectly configured on reset.	CWE-1221
BM10	Reset manager	OpenTitan SoC	Reset should follow fall of input signal within a given range of clock cycles unless input is asserted again.	Reset does not follow even after maximum clock cycles of input trigger.	CWE-1206

assertions for these benchmarks. For BM9 and BM10, the reference assertions are the same as the assertions used in the verification source code of OpenTitan. Listing 1 shows the `goldenAssert.sva` file of BM1 with the golden reference assertion at Lines 12-13.

Prompt data. Each benchmark includes a prompt data file consisting of *comment strings* describing the target assertion, *example assertions*, and *beginning strings* of the target assertion with varying amounts of details. This information is used by the prompt generator to generate multiple prompts of varying amounts of details when querying the LLM (see Section III-B). As an example, Listing 2 shows the prompt data file for BM1.

It is common practice to include comments about the property being checked when writing SVAs. For example, Listing 3 shows the SVAs of BM9 and BM10 from the OpenTitan SoC [41] where Lines 1,5 are the comments for the SVAs in Lines 2,6 respectively. It can be seen that while comments are included for the SVAs, they vary in level of detail. The comment in Line 1 does not include the name of the signals involved, while the comment in Line 2 includes the specific signal names used in the SVA. Thus, it is essential to evaluate the performance of LLMs with comment strings comprising varying levels of detail about the target assertion required. Our benchmarks include three *comment strings* to allow such evaluations as we describe below.

- **VeryBriefCom** is a comment with very few details about the target assertion. It uses common/generic names for signals instead of their actual names. For example, a signal `rst_i` would be termed `reset` in the comment. Details about timing, such as the next clock cycle and past clock cycle, are not included. It uses commonly used terminology for signal values such as `asserted` or `triggered` instead of `1` and `disabled` instead of `0`. Information about parameter values, bit widths, and exact index values for comparison are not included. For example, Line 3 of Listing 2 shows the VeryBriefCom of BM1. This comment evaluates the ability of LLM to generate assertions with minimal information about the target assertion. The LLM has to fill in the missing details, taking into account the context from the prompt.
- **BriefCom** is a brief comment that is less ambiguous than VeryBriefCom. It uses correct signal names everywhere. It specifies correct parameter values, bit-widths, and exact index values where needed. However, timing information is still not included in this comment, and common terms such as `asserted`, `triggered`, and `disabled` are still used. For example, Line 4 of Listing 2 shows the BriefCom of BM1. This comment evaluates the ability of LLM to generate assertions with most of the necessary information about the target assertion, where the LLM has to extrapolate the few missing details taking into account the available context from the prompt.
- **DetailedCom** is a detailed comment that replaces all the commonly used terms from BriefCom with exact values for the specific benchmark and includes the timing information. It also uses helping strings like appending the signal names with the word "signal" and adding the string "the value of the" when specifying the value of any signal to aid the LLM in understanding the description of the target assertion. For example, Line 5 of Listing 2 shows the DetailedCom of BM1. This comment evaluates

Listing 2. Prompt data file for BM1 showing the different values for the two components of the assertion clue, i.e., example assertion and the comment for the target assertion and the beginning of the assertion strings.

```
{
    "commentStrings": {
        "VeryBriefCom"      : "assert that the register is not changed if it is locked\n",
        "BriefCom" : "assert that the data is not changed if the lock is set\n",
        "DetailedCom" : "assert that at every positive edge of clock, the value of the data register is same as its
        ↪  value in the previous clock cycle if the value of the lock signal in the previous clock cycle is 1\n",
      },

    "examples": {
        "NoEx" : ["\n\n"],
        "TrivialEx"  : [" r_en is 0 if w_en is 1\n assert property (@(posedge clk) (w_en == 1) |-> (r_en == 0));\n\n"]
        "BasicEx"  : [" r_en is 0 if w_en is set\n assert property (@(posedge clk) ($past(w_en) == 1) |-> (r_en ==
        ↪  0));\n\n"],
        "DetailedEx"  : [" at every positive edge of clock, the value of the r_en signal is same as its value in the
        ↪  previous clock cycle if the value of the w_en signal in the previous clock cycle is 0\nassert property
        ↪  (@(posedge clk) ($past(w_en) == 0) |-> (r_en == $past(r_en)));\n\n"],
      },

    "assertionBeginning": {
        "EmptyBeg": "",
        "ShortBeg": "assert",
        "NormalBeg": "assert property (@(posedge clk)"
    }
}
```

这个部分定义了三种不同详细程度的注释字符串

这个部分提供了不同详细程度的断言例子。首先，它们为LLM设置上下文，并指导它生成断言而不是普通的设计代码。这是OpenAI在使用LLM时推荐的最佳实践。其次，这些断言帮助LLM将用户提供的目标断言描述转换为SVA

Listing 3. SVAs of BM9 and BM10 from the OpenTitan SoC with comments describing the SVAs.

```
 // FSM software reset
`ASSERT(WakeupTimerCntSwReset_A, cfg_fsm_rst_i |=>
↪  wakeup_timer_cnt_q == 0, clk_aon_i, !rst_aon_ni)

 // A fall in por_n_i leads to a fall in
 ↪  rst_por_aon_n[0].
`FALL_ASSERT(CascadePorToAon,
↪  por_n_i[rstmgr_pkg::DomainAonSel],
↪  resets_o.rst_por_aon_n[rstmgr_pkg::DomainAonSel],
↪  PorCycles, clk_aon_i)
```

the ability of LLM to generate assertions when all the information is fully specified in the form of a clear and unambiguous comment.

Each benchmark has four *example assertions*. They serve two purposes. First, they set the context for the LLM and guide it towards generating assertions rather than normal design code. This is recommended by OpenAI as a best practice when using LLMs*. Second, the assertions aid the LLM in converting the user-provided description of the target assertion into an SVA as explained below.

- **NoEx** is an empty string where no example assertion is included in the query, as shown in Line 9 of Listing 2. This example allows us to analyze if setting the context with an example assertion is needed or not when querying the LLM for assertions. As this is an empty string, all the benchmarks use the same NoEx.
- **TrivialEx** is a trivial SVA not related to any of the benchmarks. Thus, it is useful in evaluating if the LLM can generate the assertion only based on the context and without requiring any additional aid about the target assertion. All the benchmarks use the same TrivialEx, which is shown in Line 10 of Listing 2.

*https://beta.openai.com/docs/guides/code/best-practices

- **BriefEx** is a basic SVA that is a simpler version of the target assertion of the benchmark with different signal names and values. It uses the BriefCom-style comment. For example, Line 11 of Listing 2 shows the BriefEx of BM1. This comment lets us evaluate if the LLM can generate assertions when it is provided with a similar example assertion but with less complexity and a brief description of the comment. An example of this can be to use a common example for a group of hardware vulnerabilities.
- **DetailedEx** is an SVA similar to the target assertion but with different signal names and values. It uses the same comment as BriefEx. For example, Line 12 of Listing 2 shows the DetailedEx of BM1. This example evaluates if the LLM can generate the target assertion when it is given a very similar assertion along with natural language context.

The prompt data of benchmarks also include assertion "*beginnings*" (preambles) containing different amounts of context. These strings aim to help LLMs in generating the target assertion by specifying which logic to use for the assertion. Each of our benchmarks has three possible "*beginnings*" as explained below:

- **EmptyBeg** is an empty string common for all benchmarks.
- **ShortBeg** is 1-to-3 starting words of the target assertion.
- **NormalBeg** is 3-to-8 starting words of the target assertion.

Testbench. Each benchmark includes a SystemVerilog testbench that drives all the signals involved in the assertion for all possible combinations of values. We built testbenches for all the benchmarks using a single template testbench file as shown in Listing 4. We developed this template testbench manually, which parametrizes the number of signal bits involved in the assertion, *noDutSignalBits* and the base-2 logarithmic of the

Listing 4. Template testbench of our framework.

```
1  `timescale 1ns/10ps
2  module tb();
3
4    // <Parameter declarations here>
5
6    // <All signal declarations here>
7
8    localparam noDutSignalBits = <total number of signal
     ↪  bits>;
9    localparam noClocks = <number of clock cycles involved
     ↪  in assertion>;
10   localparam log2NoClocks = <log2(noClocks) ceiled to
     ↪  integer>
11   localparam CTR_WIDTH = (noDutSignalBits*noClocks) +
     ↪  log2NoClocks;
12     // + log2NoClocks to keep track of how to update
       ↪  test data
13
14   // generate clock and reset
15   initial begin
16     clk = 'b0;
17     rst = 'b1;
18     #18 rst = 'b0;
19   end
20   always #5 clk <= ~clk;
21
22   // generate tests
23   reg [CTR_WIDTH-1:0] test_data;
24   wire [noDutSignalBits-1:0] test_data_curr;
25   always @(posedge clk) begin
26     if (rst) begin
27       test_data <= 'b0;
28     end else begin
29       if (test_data == {CTR_WIDTH{ 1'b1}}) begin // stop
         ↪  since all inputs are tested
30         #5 $display("Testing done, no inputs=%d",
           ↪  test_data+1);
31         $finish;
32       end else begin
33         #5 test_data <= test_data + 1;
34       end
35     end
36   end
37
38   // <assign correct data from the counter to the
     ↪  test_data_curr>
39   // ex:  assign test_data_curr = test_data[0]
     ↪  ?test_data[(1*noDutSignalBits)+log2NoClocks +:
     ↪  noDutSignalBits]  :
40     //test_data[log2NoClocks +: noDutSignalBits];
41
42   // <assign test_data_curr to all the signals>
43
44 endmodule
```

number of clock cycles (*noClocks*) required to verify each assertion, *log2NoClocks*. Once these two parameters are set, the template testbench generates a counter of size *CTR_WIDTH* (see Line 11) and increments the value of this counter from 0 to $2^{CTR\_WIDTH} - 1$ (see Line 29). All the signal bits are driven by the MSB $noDutSignalBits \times noClocks$ bits of this counter (see Lines 38-42) while the LSB *log2NoClocks* bits keep the MSB bits unchanged for *noClocks* clock cycles. This template can be easily edited to create the testbench for any benchmark.

B. Prompt Generator

The prompt generator generates the prompt string and parameter values for each benchmark and invokes the LLMs to generate all the assertions. We automate the prompt generation process by using templates for prompt strings, as shown in Figure 2. For a given benchmark, our framework uses this

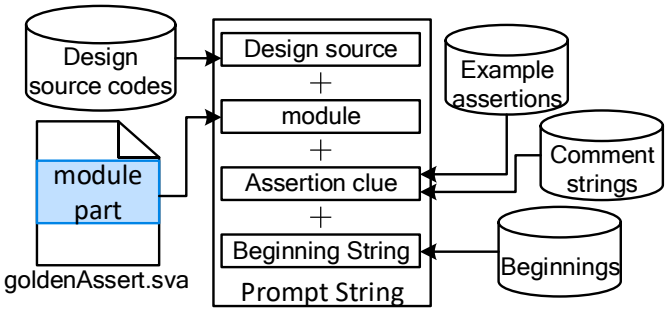

Figure 2. Template for the prompt string.

Table II
AUTOMATED SYNTAX AND TYPOGRAPHICAL FIXES USED BY OUR EVALUATION FRAMEWORK.

Fix ID	Description	Justification
R1	Remove any characters that are not ASCII	Non-ASCII characters can affect tooling
R2	Remove any characters after "endmodule"	Assertion should not span across modules
R3	Remove characters enclosed in triple quotes (""")	Code in tripe quotes is most likely a stray Python comment, and is not valid SystemVerilog syntax
R4	Add "endmodule" if not present	"endmodule" is the required keyword to end any module in SystemVerilog

template to evaluate the LLM by querying with combinations of prompt strings and analyzing the performance of the generated assertions.

The *design source* part of the prompt string consists of one of the three design source codes. The *module* part of the prompt string is generated from the `goldenAssert.sva` file of each benchmark until the location where the golden assertion is written. For BM1, the module part of the prompt string is Lines 2-9 in Listing 1. The assertion clue is formed by appending the *example assertion* and the *comment string* from the prompt data file of the benchmark. One of the three *beginning strings* of assertion is used as the last component of the prompt string. Each possible combination of (*design source*, *example assertion*, *comment string*, and *beginning*) along with the *module* part are appended to generate multiple prompts per benchmark.

In addition, the commonly-used phrase "assert that" in the prompt is replaced with another popularly-used synonym "property to check that" to evaluate the effectiveness of using synonyms in the prompt string for generating assertions. For example, Listing 5 shows a prompt string for BM1. Lines 1-17 are the *design source* code part of the prompt string. Lines 19-22 are the *module* part of the prompt string derived from `goldenAssert.sva` file of BM1, i.e., Lines 2-9 in Listing 1. Lines 24-27 comprise the assertion clue part of the prompt, where Lines 24-25 is the example assertion, and Line 27 is the comment string describing the target assertion. Note that the comment in Line 24 uses the "assert that" string while the comment in Line 27 uses the "property to check that" string. Finally, Line 28 is the *beginning string* that is used to prime the LLM into completing the code with an assertion.

Listing 5. Prompt string of BM1 showing its different components (reformatted for length).

```
`timescale 1ns/10ps
module lock_reg (
    input data_in,  output data_out,  input r_en,
    input w_en,  input lock,  input clk,  input rst);

reg data;
always @ (posedge clk) begin
    if (rst) begin
        data <= #1 0;
    end
    else begin
        if (w_en)
            data <= #1 lock ? data: data_in;
    end
end
assign data_out = r_en ? data : 'b0;
endmodule

module v_dut    (
    lock,    clk,    rst,    data  );
    input lock;    input clk;
    input rst;    input data;

 // assert that r_en is 0 if w_en is 1
 assert property (@(posedge clk) (w_en == 1) |-> (r_en
 ↪  == 0));

// property to check that at every positive edge of
↪  clock, the value of the data register is same as its
↪  value in the previous clock cycle if the value of
↪  the lock signal in the previous clock cycle is 1
 assert
```

Listing 6. Example simulation log showing assertion violation information (trimmed and reformatted for length).

```
Reading pref.tcl
# 2021.1
...
# Loading work.i_cu_assertion(fast)
# Loading work.v_dut(fast)
# run 200000us
# ** Error: Assertion error.
#    Time: 95 ns Started: 95 ns  Scope:
↪  tb.i_bind_dut_buggy File:
↪  assertion_gen_5435_7484xO.sva Line: 18
# GOLDEN: FAIL, time=  95, data=0, data_d=1, lock=1
# ** Error: Assertion error.
#    Time: 145 ns Started: 145 ns  Scope:
↪  tb.i_bind_dut_buggy File:
↪  assertion_gen_5435_7484xO.sva Line: 18
# GOLDEN: FAIL, time= 145, data=1, data_d=0, lock=1
# ** Error: Assertion error.
#    Time: 165 ns Started: 165 ns  Scope:
↪  tb.i_bind_dut_buggy File:
↪  assertion_gen_5435_7484xO.sva Line: 18
# GOLDEN: FAIL, time= 165, data=1, data_d=0, lock=1
...
# Testing done, no inputs=        32
# ** Note: $finish    :
↪  copilot_assertions//benchmarks/0/tb/tb.sv(32)
#    Time: 340 ns  Iteration: 0  Instance: /tb
# End time: 08:56:00 on Jan 19,2023, Elapsed time:
↪  0:00:02
# Errors: 7, Warnings: 0
```

C. Assertion File Generator

The assertion file generator receives all the assertions generated by the LLM from the prompt generator. It first processes these generated assertions to fix a limited set of *minor* mistakes (akin to typos) made by the LLM, as shown in Table II. These rules were derived through early qualitative analysis of the LLM outputs and reflect the "common pitfalls" made by the models, where they are straightforward errors that make otherwise-correct answers fail simulation. Since they are simple, fixes can be made automatically using lexical tooling. Further, to ensure that these repairs do not impact LLM performance, repairs are made to copies of the generated assertion, i.e., the original generated assertion remains in the list of generated assertions. We thus have *original* assertions and generated-plus-fixed assertions as *processed* assertions.

The assertion file generator creates a `generatedAssert.sva` file for the *processed* assertions by replacing the golden reference assertion in `goldenAssert.sva` with the *processed* assertions.

D. Simulator

The simulator builds SystemVerilog projects for each assertion using the SVA file generated by the assertion file generator and the corresponding testbench files. These projects are simulated using Siemens Modelsim [42] to generate the simulation log. This log has the assertion violation trace data. Listing 6 is an example simulation log. Lines 8,11,14 indicate violations of the *processed* assertion whereas Lines 9,12,15 indicate violations of the golden reference assertion.

E. Scoreboard

The scoreboard collects all simulation logs, parses the assertion violation data, and analyses the data by comparing the assertion violation data of *processed* assertions with reference golden assertion in each benchmark. The assertion is classified as "correct" if it generated violations for the same inputs as that of the golden assertion. Otherwise, they are classified as "incorrect". To highlight the number of unique assertions, the scoreboard does whitespace trimming on the generated assertions and records the number of unique *processed* assertions that are simulated correctly.

IV. Experimental Setup and Results

We ran our experiments on a 32-core, 2.6 GHz Intel Xeon workstation with 512 GB of RAM running CentOS Linux release 7.9.2009.

A. Study Overview

We use our framework to evaluate the performance of a large language model (LLM) in generating hardware security assertions. Our experiments aim to answer the two research questions RQ1 and RQ2. To this extent, we use the most popular code generation engine, OpenAI's Codex [20] (specifically, `code-davinci-002`) as our LLM, although other LLMs can be used in future work. At the time of experimentation, `code-davinci-002` engine is the most capable engine among the OpenAI engines designed to "understand" and generate code. Later we use more LLMs such as OpenAI's Codex [20] `code-cushman-001`, `codegen-2b-ft` [43], and `ChatGPT` [44] to demonstrate the scalability of our framework.

Table III
NUMBER OF ASSERTIONS GENERATED, COMPILED, SIMULATED, AND CORRECT BY THE code-davinci-002 LLM.

BM ID	# assertions generated			# assertions compiled			# assertions simulated			# correct assertions		
	Original	Processed		Original	Processed		Original	Processed		Original	Processed	
		All	Unique		All	Unique		All	Unique		All	Unique
BM1	22680	22696	16718	4767	4767	2084	4488	4488	1814	1587	1587	207
BM2	22680	22688	21137	5165	5165	3916	4401	4401	3233	603	603	203
BM3	22680	22687	20839	7985	7986	6388	7531	7532	5936	998	998	380
BM4	22680	22682	9100	12435	12435	2002	12221	12221	1866	9762	9762	914
BM5	22680	22683	18957	10277	10277	7498	10170	10170	7391	1650	1650	783
BM6	22680	22682	12532	14088	14088	4414	13925	13925	4251	2169	2169	617
BM7	22680	22694	17912	7571	7572	4325	7428	7429	4182	331	331	133
BM8	22680	22689	16996	7857	7857	3318	7566	7566	3028	1088	1088	185
BM9	22680	22685	15408	6516	6516	2768	6332	6332	2586	1266	1266	151
BM10	22680	22687	16338	5431	5431	2865	5312	5312	2746	1610	1610	198
Total	226800	226873	165937	82092	82094	39578	79374	79376	37033	21064	21064	3771

B. Evaluation Setup

We use the ten benchmarks introduced in Section III-A to evaluate the `code-davinci-002` LLM. While the benchmarks BM1 and BM2 are manually crafted "toy" designs, the rest of the benchmarks represent a wide range of real-world designs, vulnerabilities, and common weakness enumeration (CWE) categories.

LLM configuration: The `code-davinci-002` LLM is configured to generate a maximum of 256 tokens, as the largest golden reference assertion among our benchmarks only needs about 160 tokens. For stop tokens, "endmodule" is used as the sva file should not have any logic after this keyword (i.e., the assertion should end here). The "top P" and "presence penalty" parameters are set to their default values of 1 and 0, and "n" is set to generate 10 assertions for every query.

Each benchmark has three *comment strings* and four *example assertions*. Combined with the two different synonym words that can be used for each *comment string* and three of the four *example assertions*, they result in six variants of *comment strings* and seven variants of *example assertions*. These combined with the three *design source* strings and three *assertion beginning* strings result in $(6 \times 7 \times 3 \times 3) = 378$ unique query strings (prompts)..

We vary the "temperature" and "frequency penalty" parameters with the values $\{0.4, 0.9\}$, and $\{0, 0.5, 1\}$, respectively, in our experiments. Thus, for each prompt string, the framework generates six queries, one with each possible combination of temperature and frequency penalty covering a range of different parameter configurations of Codex resulting in **2268 queries to the engine for each benchmark**.

While these values are used in our evaluations, one can configure the framework to use any other values by simply updating its configuration file. Our framework is automated, starting from the generation of prompts to evaluating the performance of the LLM engine. We use Siemens Modelsim [42] as the simulation tool.

Next, we analyze the performance of the `code-davinci-002` LLM on our ten benchmarks to answer the research questions RQ1 and RQ2, while evaluating multiple other LLMs to demonstrate the scalability of our framework.

C. RQ1: Can LLMs generate hardware assertions?

Table III presents the performance of the `code-davinci-002` LLM over the benchmark suite. For each benchmark, the LLM is queried 2268 times with $n = 10$, as discussed in Section IV-B, thus generating 22680 assertions. These are then processed (see Section III-C) and deduplicated, with values presented in the "# assertions generated" column. For each column, the "Original" assertions are generated by the LLM, while the "Processed" assertions include both the assertions generated by the LLM and the repaired copies of the assertions generated by the assertion file generator. Column "All" represents the total number of processed assertions, while column "Unique" represents the unique number of assertions. The number of unique assertions is the number of unique processed assertions after filtering out the duplicate whitespaces.

From here, we then check the initial validity of each assertion. The first check involves compilation by Modelsim. These must be checked as some assertions will have invalid syntax or refer to invalid signals or parameter names or values. The successful compilations are presented in the "# assertions compiled" column.

The next check involves ensuring the assertions may be functionally simulated. This means that the simulations reach the end of their respective testbenches after all the possible combinations of input values are tested. This column is important as some assertions may get stuck during the simulation (for example, by using SystemVerilog constructs like `$stop`). This value is presented in "# assertions simulated".

Finally, the number of assertions which are correct is presented in the "# correct assertions" column. This is the number of assertions that trigger violations for same combinations of input values as that of the golden reference assertion.

It can be seen from the correct assertions column that the LLM can successfully generate correct assertions for all 10 benchmarks. However, on average, only 9.29% of the generated assertions are correct. Hence, a random query to

the LLM from our framework has a very low probability of generating the correct assertion. Also, it can be seen from the number of unique compiled and unique correct assertions columns that the LLM generates a wide range of assertions even when the prompt string has details corresponding to a specific assertion.

Takeaway. Results from Table III indicate that LLMs can indeed generate hardware security assertions. This being said, in addition to the correct assertions, it generates many incorrect and non-functional/non-compiling assertions. Understanding what might encourage the LLM to function reliably is the goal of RQ2.

D. RQ2: How does the LLM perform for different types of prompts?

In this section, we analyze Codex `code-davinci-002` LLM's performance when varying different components of the query: (i) *example assertions* and *comment strings*, (ii) *Design source codes* and comment strings, (iii) Assertion *beginning strings*, (iv) *Synonym strings*, and (v) temperature and frequency penalty values.

1) Example assertions and comment strings: The *example assertion* and the *comment string* are important elements of the prompt string as they provide the context for the LLM prompt, providing 'hints' about the target assertion. Figure 3 shows the percentage of correct assertions generated by the LLM when using all possible combinations of the *example assertions* and *comment strings* on all benchmarks. The following inferences can be made:

- Combinations with *NoEx* did not perform well on any of the benchmarks. This is likely due to the lack of context for the target assertion informing the LLM to generate an assertion rather than normal design code.
- Combinations such as *TrivialEx* with *BriefCom*, *BasicEx* with *VeryBriefCom* did not perform well on any benchmark except for BM3 and BM4. This shows that not all queries perform similarly on all the benchmarks.

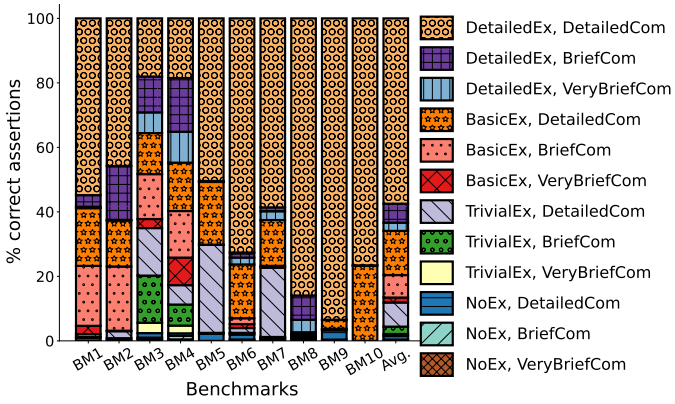

Figure 3. Comparing percentage of correct assertions generated by the `code-davinci-002` LLM when using combinations of *example assertions* and *comment strings*.

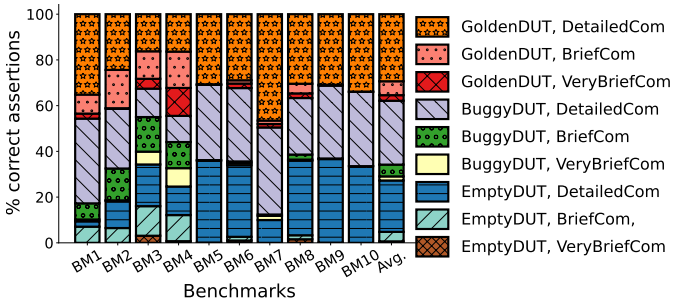

Figure 4. Comparing percentage of correct assertions generated by `code-davinci-002` LLM when using different combinations of *design source codes* and *comment strings*.

- On average, across all the benchmarks, (*DetailedEx*, *DetailedCom*), (*BasicEx*, *DetailedCom*), and (*TrivialEx*, *DetailedCom*) pairs are the top three performers. This is because *DetailedCom* elaborates all the details of the target assertion such as correct signal names and timing information allowing the engine to generate the correct assertion.

Takeaway. The results from Figure 3 indicate that of the 12 combinations of the prompt strings, the best three combinations result in 80% of the correct assertions. This reinforces the need to identify and use suitable prompt strings to maximize the correctness of the generated assertions. In general, the greater the detail in prompt, the greater the proportion of successfully generated assertions.

2) Design source codes and comment strings: The LLM infers the information about the target assertion such as signal names and timing information from the *design source code* and the *comment string* in the prompt. The information in *design source code* is not easy to infer since it is mixed with other design logic and is distributed across multiple modules. On the other hand, the information in the *comment string* is easy to infer since it clearly describes the properties of the target assertion and is located near the assertion. Hence, the information provided through *comment string* is more useful to the LLM in generating a correct assertion. However, comment stings need to be input by the user unlike the *design source code*. Hence it is essential to evaluate the dependence of LLM on the information in the *design source code* and the *comment string* parts of the prompt string. Figure 4 shows how the LLM performs when using three different types of *design source codes* and *comment strings*. The following inferences can be drawn:

- Combinations with *DetailedCom* perform well in most of the benchmarks even with *EmptyDUT*. This is because all the information required to generate the target assertion is provided in the *DetailedCom*.
- *EmptyDUT* performed poorly when used with *VeryBriefCom* and *BriefCom* because the complete information about the target assertion is neither in the *design source code* nor in the *comment string*.
- *GoldenDUT* and *BuggyDUT* prompts are able to generate

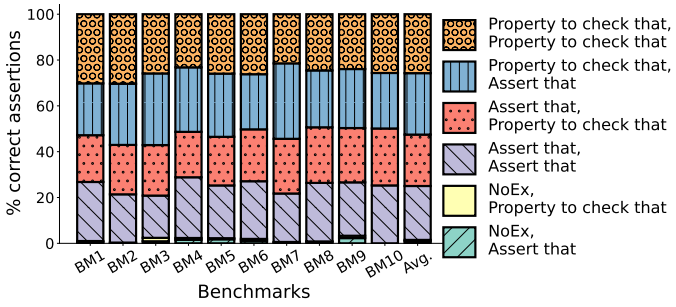

Figure 5. Comparison of the percentage of correct assertions generated by the `code-davinci-002` LLM when using different combinations of *synonym strings*.

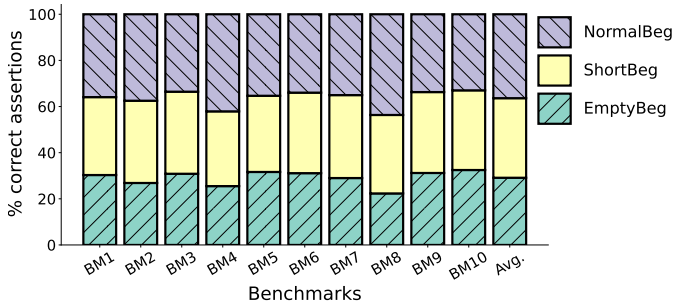

Figure 6. Comparison of the percentage of correct assertions generated by the `code-davinci-002` LLM when using different combinations of assertion *beginning* strings.

correct assertions even when using *BriefCom* because the LLM infers the information about the target assertion from the *design source files*. But not all benchmarks perform well with these combinations because it is not trivial to infer information from the *design source files* in complex designs.

- On average, *GoldenDUT* prompts performed the best as they contain correct information about the benchmark.

Takeaway. The results from Figure 4 indicate that either *design source code* or *comment string* should contain the information about the benchmark design and the target assertion to generate correct assertions. While providing details through *comment string* is more effective, LLMs should be trained to utilize the *design source code* in generating the assertions as that minimizes the user input.

3) Synonym strings.: The comments in *example assertions* and *comment strings* begin with different synonyms such as *assert that* and *Property to check that*. Figure 5 shows the correctness of generated assertions when using different combinations of *synonym strings* in the comments part of the prompt. Here, *NoEx* indicates that the example did not use any of the synonyms because no example was used in these prompts. Hence there are only synonyms used by the *comment strings*. The following inferences can be made from this comparison:

- All the benchmarks have a similar distribution of the results, indicating that the synonyms used in the example and comment string are independent of the assertion.
- The combinations with *NoEx* did not perform well. This is not because of the synonym used but because the prompts with *NoEx* generated ¡ 2% of correct assertions on average (see Figure 3).
- Excluding *NoEx* combinations, the remaining synonym combinations performed equally well in generating the assertions. However, the combinations (*Property to check that*, *Assert that*) and (*Property to check that*, *Property to check that*) performed better than the other combinations.

Takeaway. The results from Figure 5 indicate that as long as the comment is included in the prompt, using different synonyms has little impact on the correctness of the assertions generated.

Assertion beginnings. Providing the beginning of the assertion sets the context for the LLM engine in generating the assertions. We evaluated all the benchmarks with the three different assertion beginning strings as discussed in Section III-A. Figure 6 plots these results. The following inferences can be made from this comparison:

- All three assertion beginning strings performed similarly on most of the benchmarks. This shows that adding the beginning of an assertion in the prompt has little impact on the assertions generated. A major reason for this is that the prompts have *example assertions* and *comment strings* detailing the assertion that sets the context for the LLM.
- On benchmarks like BM4 and BM8, providing the beginning of an assertion improved correctness of the generated assertions. This is because these assertions require SystemVerilog `generate` construct. *NormalBeg* and *ShortBeg* both provide `genvar` keyword that hints the LLM engine to use `generate` construct to generate correct assertions.
- On average, across all the benchmarks, all three assertion beginning strings equally with *NormalBeg* perform best due to its better performance on BM4 and BM8.

Takeaway. The results from Figure 5 indicate for most benchmarks, *EmptyBeg*, i.e., the empty assertion beginning string is sufficient in generating correct assertions. For assertions requiring constructs other than the trivial `assert` construct, providing the beginning keyword of the construct such as `genvar` improves the probability of generating correct assertion.

4) Temperature and frequency penalty values.: The assertions generated by the LLM depend on the values of various hyper-parameters used apart from the prompt string itself. We vary the *temperature* and the *frequency penalty* in our experiments to analyze their impact on the assertions generated for different benchmarks. Figure 7 plots these results. The following inferences can be made from this comparison:

- The performance of the LLM engine using different parameters is benchmark dependent. For example, in the case of BM3, a temperature of 0.9 generated ¿ 75% of

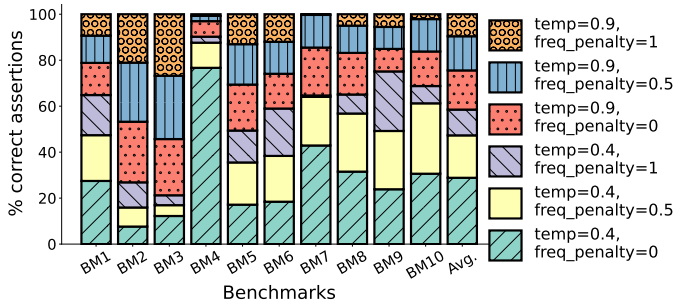

Figure 7. Comparison of the percentage of correct assertions generated by the `code-davinci-002` LLM when using different combinations of *temperature* and *frequency penalty* values.

the correct assertions, while in BM4, the same temerature value generated ¡ 10% of the correct assertions.

- On average across all the benchmarks, both temperatures performed equally well whereas frequency penalty value of 0 performed better than 0.5 and 1.

Takeaway. The results from Figure 5 indicate that while both the temperature values performed equally well, the frequency penalty value of 0 performed better than 0.5 and 1.

E. Scalability To Other LLM Engines

To demonstrate the scalability of our framework, we evaluated three different LLMs along with *code-davinci-002* with our benchmark suite using the best-performing query on *code-davinci-002* LLM (see Section IV-D). This query uses *GoldenDUT* as the *design source code*, *DetailedEx* as the *assertion example* with *property to check that* synonym, *DetailedCom* as the *comment string* with *assert that* synonym, *NormalBeg* as the assertion *beginning* string. The temperature value is 0.4 and frequency penalty value is 0. The three other LLMs are: (i) OpenAI Codex [20] *code-cushman-001*, (ii) codegen-2b-ft [43], and (iii) ChatGPT [44] (Jan 9 version). Table IV shows the results of this evaluation demonstrating that our framework can easily scale to different LLMs. It can be seen that *code-davinci-002* correctly generated assertions for nine of the 10 benchmarks with this single query while a simialar engine, *code-cushman-001* generated correct assertions for seven benchmarks. *codegen-2b-ft* only generated assertions for four benchmarks. The reason why ChatGPT generated corrections for only five benchmarks is because it tries to explain the code in the query and provide generic suggestions to create assertions for that benchmark rather than generating the target assertion.

F. Other Observations

In this section, we discuss more observations from experiments.

Multiple correct assertions are possible for a benchmark even though the assertions target specific vulnerabilities in the design. For example, the assertion for BM2 should make sure that the traffic light changes to RED only from yellow. Listing 7 shows correct assertions generated by Codex when queried. Our scoreboard qualifies the presence of duplicate

Table IV
EVALUATION USING DIFFERENT LLM ENGINES.

	code-davinci-002	code-cushman-001	codegen-2b-ft	ChatGPT
BM1	✓	✓	✓	✓
BM2	✓	✗	✗	✓
BM3	✗	✓	✗	✗
BM4	✓	✗	✓	✓
BM5	✓	✓	✓	✗
BM6	✓	✓	✓	✓
BM7	✓	✗	✗	✗
BM8	✓	✓	✗	✓
BM9	✓	✓	✗	✗
BM10	✓	✓	✗	✗

Listing 7. Correct assertions for BM2 generated by Codex.

```
1  // generated assertion 1
2  assert property (@(posedge clk)  (signal == RED) |->
   ↪  $past(signal) == RED | $past(signal) == YELLOW);
3
4  // generated assertion 2
5  assert property (@(posedge clk)  ( (signal[1:0] == RED)
   ↪  |-> ( ($past(signal[1:0]) == RED) |
   ↪  ($past(signal[1:0]) == YELLOW) ) )) ;
6
7  // generated assertion 3
8  assert property (@(posedge clk)  (signal == RED) |->
   ↪  ($past(signal) == YELLOW | $past(signal) == RED));
9
10 // generated assertion 4
11  always @ (posedge clk)
12   if(signal == RED)
13     assert($past(signal) == RED || $past(signal) ==
       ↪  YELLOW);
```

assertions by filtering out similar assertions and determining the unique assertions as listed in Table III.

Multiple generated assertions. A single query to the LLM can result in the generation of multiple assertions. For instance, Listing 8 shows the responses generated by Codex that consist of more than one assertion. Having more than one assertion in the response of LLM can lead to the correctness or incorrectness of the assertion. The response at Lines 1-3 consists of multiple essential assertions, i.e., the response is correct because of the presence of multiple assertions. Whereas in the case of the response at Lines 15 to 19, the generated response is incorrect even though it contains the target assertion (Line 16) because the assertion at Line 19 is incorrect, resulting in incorrect overall behavior in simulation. Also in the case of responses such as Lines 5-14, the additional assertion (Line 13) does not impact the correctness of the assertion; it is a subset of other assertions in the response or is triggered in the design.

The multiple assertions in the response of LLM indicate that a semi-colon cannot be used as the stop sequence; the semi-colon limits the LLM to generating only one assertion.

V. DISCUSSION AND LIMITATIONS

A. Completeness of Benchmark Suite

Our experimental work featured ten realistic golden reference assertion benchmarks (Table I) that we used to evaluate the correctness of the LLM-generated assertions. While our benchmarks represent a range of properties, future work can

Listing 8. Responses generated by OpenAI Codex consisting of multiple assertions per response.

```
// response for BM8 -- the multiple assertions present are required for correctness
 assert property (@(posedge clk) $past(rst_i) |-> (reglk_mem[0] == {BIT_WIDTH{1'b1}}));
 assert property (@(posedge clk) $past(rst_i) |-> (reglk_mem[1] == {BIT_WIDTH{1'b1}}));

// response for BM1 -- consists of correct but redundant assertions

assert property (@(posedge clk)  ($past(lock) == 1) |-> ($rose(data) == 0));

// property to check that the data is not changed if the lock is set
 assert property (@(posedge clk) ($past(lock) == 1) |-> ($fell(data) == 0));

// property to check that the data is not changed if the lock is set
 assert property (@(posedge clk) ($past(lock) == 1) |-> ($stable(data) == 1));

// response for BM3 -- consists of unwanted assertion leading to incorrect response
assert property (@(posedge clk) (data == RED) |-> ($past(data) == RED) || ($past(data) == YELLOW));

// assert that the WALK signal is always preceeded by the RED or the YELLOW signals
    assert property (@(posedge clk) (walk == 1'b1) |-> ($past(data) == RED ) || ($past(data) = YELLOW));
```

consider an even broader range of CWE types and vulnerabilities that were not explored in this paper. For all our benchmarks, we created the different levels of comment strings based on our human judgment of their relative complexity/detail. Except for BM9 and BM10, we also designed the reference assertions based on our assessment of the example vulnerabilities/CWE types. As such, our reference assertions are only one of several possible ways to capture the desired security property related to each vulnerability/CWE. Our comment strings similarly represent only a small subset of possible ways to express intent in the prompt to the LLM – other ways of constructing the prompt could be explored in future work.

B. Simulation TestBench

Our method for evaluating the generated assertions involved simulating both the golden reference assertion and the generated ones with all signal value combinations for signals involved in the assertion (i.e., an exhaustive test) and seeing if the behavior matched (Section III-A). To make this testing more practical, we parameterized the signal widths used in the assertions so that we could simulate signals using different widths; for instance, 32-bit signals can be treated as 2-bit signals, so we needed to simulate only 2^2 possible values of that signal instead of 2^{32} values, with the assumption that the observed match or mismatch in behavior between reference and generated assertion holds without loss of generality. Furthermore, as our reference assertions are not the definitive way to express the desired security properties, we note that it is possible that a generated assertion could actually be a better representation of the security intent but is marked as incorrect by our framework due to simple criteria of mismatching our hand-crafted references.

Some assertions involve checking signal values at current and previous clock cycles. As an example, assume that an assertion involves 10 binary signals. There are 2^{10} values in a given cycle – to check an assertion in a current and previous cycle, we need to test $2^{10} \times 2^{10}$ combinations. For assertions that track values across more cycles, we need to test more combinations. For our benchmarks, we manually ascertained how many clock cycles are required for the reference assertion in the design of the corresponding testbench.

In some instances, simulation of the assertions takes longer than expected time. All reference assertions are exhaustively tested in less than 100 seconds of wall-clock time. So we set a timeout of 1000 seconds of wall-clock time when the generated assertion has constructs that would cause the simulation to "hang" (e.g., SystemVerilog's $stop statement). In such situations, we still marked an assertion as correct if its functional behavior matched the reference assertion.

C. Use-cases of our Framework

We emphasize that our framework is designed primarily to evaluate LLMs in generating hardware security assertions. While we used our framework on OpenAI's `code-davinci-002`, other models could be used in the future. We envision that our framework can be used to help train more specialized models for generating assertions. With regards to using our framework to generate assertions in a production environment, however, several practical challenges require further investigation. Our results show that some level of detail is required in the comments capturing security intent, which suggests that our framework as-is is not usable without some amount of security expertise. We note that related works like UNDINE [36] could assist in this regard by mining for invariant properties. Preparing the prompt involves selecting the appropriate "module part," – identifying which signals are relevant and what additional context should be provided in the prompt. This remains an open area of inquiry. Our work focused on concurrent assertions in separate sva files keeping the design files unmodified. Generating assertions like immediate assertions that are written into the hardware designs themselves could be future work.

For this process to be used in practice, evaluating whether or not the assertions (a) analyze the desired security properties (i.e., are relevant) and (b) accurately assess this property (i.e., are correct) will require additional expertise, especially when golden reference assertions are not present.

VI. Conclusion

This paper provides the first insights on using LLMs such as OpenAI's Codex for the automatic creation of Security Assertions for Hardware. For this, we contributed a new pipeline for evaluation of LLMs for this task, as well as provide a benchmark suite of scenarios and golden reference assertions. Using this for the `code-davinci-002` LLM, we generated 75,600 assertions under a variety of environmental

conditions and with differing amounts of context, and found that in general LLMs can indeed perform this task, on average generating correct assertions 4.53% of the time, with the more detailed comments and context tending to generate correct assertions most effectively. Although the overall success rate is small, these results indicate a valid proof-of-concept for the approach and show that LLMs can function in this area. Future work will examine how additional LLMs can be fine-tuned for this purpose for improved performance and determining new methods for quickly evaluating assertion correctness.

ACKNOWLEDGEMENT

We thank the Hack@DAC organizers, including Dr. Ahmad-Reza Sadeghi and Intel, for open-souring the buggy designs of the competition. We also thank the anonymous reviewers for their comments and the TAMU HRPC for their support.

REFERENCES

[1] M. Rostami, F. Koushanfar, and R. Karri, "A Primer on Hardware Security: Models, Methods, and Metrics," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1283–1295, Aug. 2014. [Online]. Available: http://ieeexplore.ieee.org/document/6860363/

[2] K. Xiao *et al.*, "Hardware Trojans: Lessons Learned after One Decade of Research," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 22, no. 1, pp. 6:1–6:23, May 2016. [Online]. Available: http://doi.org/10.1145/2906147

[3] A. Chakraborty *et al.*, "Keynote: A Disquisition on Logic Locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 1952–1972, Oct. 2020, conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[4] K. Basu *et al.*, "CAD-Base: An Attack Vector into the Electronics Supply Chain," *ACM Transactions on Design Automation of Electronic Systems*, vol. 24, no. 4, pp. 38:1–38:30, Apr. 2019. [Online]. Available: https://doi.org/10.1145/3315574

[5] P. Kocher *et al.*, "Spectre Attacks: Exploiting Speculative Execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, pp. 1–19, iSSN: 2375-1207.

[6] M. Lipp *et al.*, "Meltdown: Reading Kernel Memory from User Space," *Communications of the ACM*, vol. 63, no. 6, pp. 46–56, 2020.

[7] Y. Kim *et al.*, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, Jun. 2014, pp. 361–372, iSSN: 1063-6897.

[8] T. S. Tan and B. A. Rosdi, "Verilog HDL Simulator Technology: A Survey," *Journal of Electronic Testing*, vol. 30, no. 3, pp. 255–269, Jun. 2014. [Online]. Available: https://doi.org/10.1007/s10836-014-5449-5

[9] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: a survey," *ACM Transactions on Design Automation of Electronic Systems*, vol. 4, no. 2, pp. 123–193, Apr. 1999. [Online]. Available: https://doi.org/10.1145/307988.307989

[10] J. Rajendran, V. Vedula, and R. Karri, "Detecting Malicious Modifications of Data in Third-Party Intellectual Property Cores," *Proceedings of the 52nd Annual Design Automation Conference*, pp. 1–6, 2015.

[11] T. Trippel *et al.*, "Fuzzing Hardware Like Software," *USENIX Security Symposium*, pp. 3237–3254, 2022.

[12] R. Kande *et al.*, "TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities," *USENIX Security Symposium*, pp. 3219–3236, 2022.

[13] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register transfer level information flow tracking for provably secure hardware design," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, Mar. 2017, pp. 1691–1696, iSSN: 1558-1101.

[14] C. Chen *et al.*, "HyPFuzz: Formal-Assisted Processor Fuzzing," *arXiv preprint arXiv:2304.02485*, 2023.

[15] H. Witharana, Y. Lyu, S. Charles, and P. Mishra, "A Survey on Assertion-based Hardware Verification," *ACM Computing Surveys*, vol. 54, no. 11s, pp. 225:1–225:33, Sep. 2022. [Online]. Available: https://doi.org/10.1145/3510578

[16] Z. Ren and H. Al-Asaad, "Overview of assertion-based verification and its applications," in *International Conference on Embedded Systems, Cyber-physical Systems, & Applications (ESCS)*. CSREA Press, 2016.

[17] S. Vasudevan *et al.*, "GoldMine: Automatic assertion generation using data mining and static analysis," in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, Mar. 2010, pp. 626–629, iSSN: 1558-1101.

[18] S. Hertz, D. Sheridan, and S. Vasudevan, "Mining Hardware Assertions With Guidance From Static Analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 6, pp. 952–965, Jun. 2013, conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[19] H. Witharana, A. Jayasena, A. Whigham, and P. Mishra, "Automated Generation of Security Assertions for RTL Models," *ACM Journal on Emerging Technologies in Computing Systems*, Nov. 2022, just Accepted. [Online]. Available: https://doi.org/10.1145/3565801

[20] M. Chen *et al.*, "Evaluating Large Language Models Trained on Code," Jul. 2021, arXiv:2107.03374 [cs]. [Online]. Available: http://arxiv.org/abs/2107.03374

[21] G. Dessouky *et al.*, "Hardfails: Insights into Software-Exploitable Hardware Bugs," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. Santa Clara, CA, USA: USENIX Association, 2019, pp. 213–230.

[22] R. Mihalcea, H. Liu, and H. Lieberman, "NLP (Natural Language Processing) for NLP (Natural Language Programming)," in *Computational Linguistics and Intelligent Text Processing*, A. Gelbukh, Ed. Springer Berlin Heidelberg, 2006, pp. 319–330.

[23] A. Radford *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[24] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[25] A. Vaswani *et al.*, "Attention is All you Need," in *Advances in Neural Information Processing Systems*, vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[26] P. Gage, "A New Algorithm for Data Compression," *C Users Journal*, vol. 12, no. 2, pp. 23–38, Feb. 1994.

[27] GitHub, "GitHub Copilot · Your AI pair programmer." [Online]. Available: https://copilot.github.com/

[28] H. Pearce *et al.*, "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*, May 2022, pp. 754–768, iSSN: 2375-1207.

[29] ——, "Examining Zero-Shot Vulnerability Repair with Large Language Models," Aug. 2022, arXiv:2112.02125 [cs]. [Online]. Available: http://arxiv.org/abs/2112.02125

[30] M. Shoeybi *et al.*, "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism," Mar. 2020, arXiv:1909.08053 [cs]. [Online]. Available: http://arxiv.org/abs/1909.08053

[31] E. Nijkamp *et al.*, "A Conversational Paradigm for Program Synthesis," Mar. 2022, arXiv:2203.13474 [cs]. [Online]. Available: http://arxiv.org/abs/2203.13474

[32] H. Pearce, B. Tan, and R. Karri, "DAVE: Deriving Automatically Verilog from English," in *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*. Virtual Event Iceland: ACM, Nov. 2020, pp. 27–32. [Online]. Available: https://dl.acm.org/doi/10.1145/3380446.3430634

[33] W. Zhong, C. Li, J. Ge, and B. Luo, "Neural Program Repair : Systems, Challenges and Solutions," in *13th Asia-Pacific Symposium on Internetware*. Hohhot China: ACM, Jun. 2022, pp. 96–106. [Online]. Available: https://dl.acm.org/doi/10.1145/3545258.3545268

[34] H. Pearce *et al.*, "Can OpenAI Codex and Other Large Language Models Help Us Fix Security Bugs?" *arXiv:2112.02125 [cs]*, Apr. 2022, arXiv: 2112.02125. [Online]. Available: http://arxiv.org/abs/2112.02125

[35] C. Wang, Y. Cai, Q. Zhou, and H. Wang, "ASAX: Automatic security assertion extraction for detecting Hardware Trojans," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2018, pp. 84–89, iSSN: 2153-697X.

[36] C. Deutschbein and C. Sturton, "Mining Security Critical Linear Temporal Logic Specifications for Processors," in *2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, Dec. 2018, pp. 18–23, iSSN: 2332-5674.

[37] C. B. Harris and I. G. Harris, "GLAsT: Learning formal grammars to translate natural language specifications into hardware assertions," in *Design, Automation Test in Europe Conf. Exhibition (DATE)*, 2016, pp. 966–971.

[38] R. Zhang and C. Sturton, "Transys: Leveraging Common Security Properties Across Hardware Designs," in *2020 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2020, pp. 1713–1727. [Online]. Available: https://ieeexplore.ieee.org/document/9152775/

[39] C. Chen *et al.*, "Trusting the Trust Anchor: Towards Detecting Cross-Layer Vulnerabilities with Hardware Fuzzing," *59th ACM/IEEE Design Automation Conference*, p. 1379–1383, 2022.

[40] A.-R. Sadeghi, J. Rajendran, and R. Kande, "Organizing The World's Largest Hardware Security Competition: Challenges, Opportunities, and Lessons Learned," *Great Lakes Symposium on VLSI*, p. 95–100, 2021.

[41] lowRISC contributors, "Open source silicon root of trust (RoT) | OpenTitan." [Online]. Available: https://opentitan.org/

[42] Siemens, "Modelsim," https://eda.sw.siemens.com/en-US/ic/modelsim/, 2021, Last accessed on 04/08/2021.

[43] S. Thakur, "Finetuned codegen-2B-Verilog model," https://huggingface.co/shailja, 2022, Last accessed on 01/05/2022.

[44] OpenAI, "ChatGPT: Optimizing Language Models for Dialogue," https://openai.com/blog/chatgpt/, 2022, Last accessed on 01/05/2022.