Fixing Hardware Security Bugs with Large Language Models

Baleegh Ahmad
New York University
ba1283@nyu.edu

Shailja Thakur
New York University
st4920@nyu.edu

Benjamin Tan
University of Calgary
benjamin.tan1@ucalgary.ca

Ramesh Karri
New York University
rkarri@nyu.edu

Hammond Pearce
New York University
hammond.pearce@nyu.edu

arXiv:2302.01215v1 [cs.CR] 2 Feb 2023

ABSTRACT

Novel AI-based code-writing Large Language Models (LLMs) such as OpenAI's Codex have demonstrated capabilities in many coding-adjacent domains. In this work we consider how LLMs maybe leveraged to automatically repair security-relevant bugs present in hardware designs. We focus on bug repair in code written in the Hardware Description Language Verilog. For this study we build a corpus of domain-representative hardware security bugs. We then design and implement a framework to quantitatively evaluate the performance of any LLM tasked with fixing the specified bugs. The framework supports design space exploration of prompts (i.e., prompt engineering) and identifying the best parameters for the LLM. We show that an ensemble of LLMs can repair all ten of our benchmarks. This ensemble outperforms the state-of-the-art Cirfix hardware bug repair tool on its own suite of bugs. These results show that LLMs can repair hardware security bugs and the framework is an important step towards the ultimate goal of an automated end-to-end bug repair framework.

CCS CONCEPTS

• **Hardware** → **Hardware description languages and compilation**; • **Security and privacy** → **Hardware security implementation**; • **Computing methodologies** → *Natural language processing*.

1 INTRODUCTION

'Bugs' are inevitable when writing large quantities of code. Fixing them is laborious: automated tools are thus designed and employed to both identify bugs and then patch and repair them [9, 23]. While considerable effort has explored software repair, for Hardware Design Languages (HDLs), the state of the art is less mature.

In this study, we focus on repairing security-relevant hardware bugs. While linters [25, 49] and formal verification tools [2, 12] cover a large proportion of functional bugs, fewer tools cover hardware security bugs. Although formal verification tools like Synopsys FSV can be used for security verification in the design process, they have limited success [18]. Unlike software bugs, security bugs in hardware are more problematic because they cannot be patched once the chip is fabricated; this is especially concerning as hardware is typically the root of trust for a system [42]. With the ever-growing complexity of modern processors, software-exploitable hardware bugs are becoming common and pernicious [26, 28]. This has resulted in the exploration of many techniques such as fuzzing [46, 48], information flow tracking [7, 33, 52], unique program execution checking [21] and static analysis [5, 11]. However, very few techniques

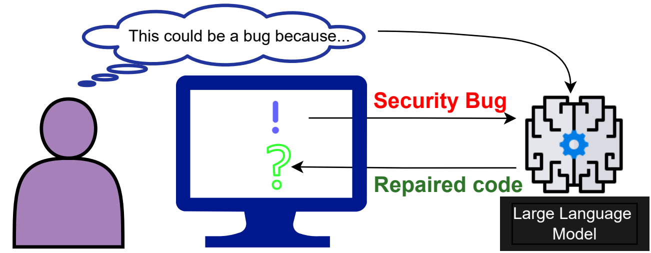

Figure 1: LLMs can suggest repairs to designers.

address the automated repair of hardware bugs. The recently proposed Cirfix [6] develops automatic repair of functional hardware bugs and, to the best of our knowledge, is the only relevant effort in this context thus far. Further efforts need to be made to support the automated repair of functional and security bugs in hardware.

Large Language Models (LLMs) are neural networks trained over millions of lines of text and code [13]. LLMs that are fine-tuned over open-source code repositories can generate code, where a user "prompts" the model with some text (e.g., code and comments) to guide the code generation. In contrast to previously proposed code repair techniques that involve mutation, repeated checks against an "oracle," or source code templates, we propose that an LLM trained on code and natural language could potentially generate fixes, given an appropriate prompt. As LLMs are exposed to a wide variety of code examples during training, they should be able to assist designers in fixing bugs in different types of hardware designs and styles, with natural language guidance. In prior work [38, 45], LLMs have been used to generate functional Verilog code. Machine learning-based techniques such as Neural Machine Translation [47] and pre-trained transformers [19] are explored in the software domain for bug fixes. Pearce et al. [37] use this approach to repair two scenarios of security weaknesses in Verilog code.

Thus, in this work, we investigate the use of LLMs to generate repairs for hardware security bugs. We study the performance of OpenAI Codex and CodeGen LLMs on instances of hardware security bugs. We offer insights into how best to use LLMs for successful repairs. An RTL designer can spot a security weakness and the LLM can help to find a fix as shown in Figure 1. Our contributions are as follows:

- Curating a benchmark of hardware security bugs and their corresponding designs. These are open-sourced at [41].

https://t.me/learningnets

- Automated framework for using LLMs to generate repairs and evaluate them. We make the framework and artifacts produced in this study available [41].
- Automated end-to-end solution to detect, repair and evaluate repairs for certain bugs utilizing static analysis scanners from prior related work [5].
- Exploration of different LLMs and their parameters to suggest how best to use LLMs in hardware bug repair. These are posed as research questions answered in Section 5.

2 BACKGROUND AND RELATED WORK

Our work borrows ideas from software domain and applies them to the area of hardware design. Since this is not very common, in this section we present some over-arching concepts that better help understand our implementation.

2.1 Code Repair

Software code repair techniques continue to evolve (interested readers can see the living review by Monperrus [32], which contains an ever-growing list of automated repair tools and techniques). Generally, techniques try to fix errors through the use of program mutations and repair templates paired with tests to validate any changes [27, 50, 51]. Feedback loops are constructed with a reference implementation to guide the repair process [30, 43]. Other domain-specific tools may also be built to deal with particular areas like build scripts, web, software models, etc.

Security bugs are critical bug types that can lead to vulnerable systems. They can be more difficult to detect and repair than functional bugs, which can be detected by classical testing. Proving the presence or absence of a security bug is challenging. This has led to more 'creative' kinds of bug repair, including AI-based machine-learning techniques such as neural transfer learning [14] and example-based approaches [31, 53]. ML-based approaches involve memorization and generalization capabilities of neural networks, allowing a greater ability to suggest repairs for "unseen" code. The example-based approaches start off with a dataset consisting of pairs of bugs and their repairs. Then, matching algorithms are applied to spot the best repair candidate from the dataset. Efforts in repair are also explored in other domains like recompilable decompiled code [40].

For digital hardware design, the recently proposed CirFix [6] attempts to localize bugs in RTL designs and then repair them. The researchers provide the benchmarks they develop for their study, allowing us to apply our methods to compare results. While it is the closest work, there are some fundamental differences in the approaches which limit direct comparisons. These differences are described in Table 1. CirFix performs both localization/identification of the bug and the repair. These two parts can be examined independently, e.g., Tarsel [52] uses hardware-specific timing information and the program spectrum and captures the changes of executed statements to locate faults effectively. Tarsel outperforms CirFix on CirFix's benchmarks as a fault localizer. In our work, we focus on the *repair* aspect. Our repair approach has the advantage that an oracle is not needed. While CirFix instruments an oracle to use the correct outputs to guide repairs, LLMs rely on the many examples of RTL code from training to produce a correct version of the buggy code. We compare our framework's performance with CirFix and discuss it in Section 5.6.

Table 1: Comparison with CirFix's approach

CirFix [6]	LLMs (e.g., this study)
Localization and repair	Repair only (assumes location)
Oracle-guided	No oracle needed
Uses repair templates and operators	Uses instructions
Iterative process	One shot

2.2 Bugs in Register Transfer Level design

Register Transfer Level (RTL) designs, typically coded in Hardware Description Languages (HDLs) such as Verilog, are high-level behavioral descriptions of hardware circuits specifying how data is transformed, transferred, and stored. RTL logic features two types of elements, sequential and combinational. Sequential elements (e.g., registers, counters, RAMs) tend to synchronize the circuit according to clock edges and retain values using memory components. Combinational logic (e.g., simple combinations of gates) change their outputs instantaneously according to the inputs. Whereas software code describes programs that will be executed from beginning to end, RTL specified in HDL describes hardware designs to be implemented. As hardware, components run independently in parallel.

Like software, hardware designs have security bugs. By definition, RTL is insecure if the security objectives of the circuit are unmet. These may include confidentiality and integrity requirements [39]. Confidentiality is violated if data that should not be seen/read under certain conditions is exposed. For example, improper memory protection allows encryption keys to be read by user code. Integrity is violated if data that should not be modifiable under certain conditions is modifiable. For example, user code can write into registers that specify the access control policy. Secure computation is a concern, and the synthesis and optimization of secure circuits starts with the description of designs with HDLs [16]. Verisketch [8] defines a synthesis language to implement timing-sensitive information flow properties to generate secure RTL.

2.3 Static Analysis

Static Analysis of code involves breaking down the code into its syntactic and lexical elements and exploring this information without simulating/compiling the code. This gives a lot of useful information, primarily in the form of an Abstract Syntax Tree (AST), which contains the variables, signals, operators, keywords, function definitions, parameters, and many other elements. Many tools have utilized static techniques in repair [10, 20]. Static analysis is helpful for bug detection and repair as it can be done in the early stages of development. This is particularly beneficial in the hardware domain as once the RTL is synthesized and fabricated into a circuit in silicon, patches are not possible, and the cost of fixing the issue increases exponentially.

https://t.me/learningnets

2.4 Common Weakness Enumerations

MITRE [15] is a not-for-profit that works with academia and industry to come up with a list of Common Weakness Enumerations (CWEs) that represent categories of vulnerabilities in hardware and software. A weakness is an element in a digital product's software, firmware, hardware, or service that can be exploited for malicious purposes. The CWE list provides a general taxonomy and categorization of these elements that allow a common language to be used for discussion. It helps developers and researchers search for the existence of these weaknesses in their designs and compare various tools they use to detect vulnerabilities in their designs and products. In this work, we address a few CWEs that our designs contain. We identify a CWE that best describes the bug.

1234: Hardware Internal or Debug Modes Allow Override of Locks. System configuration controls, e.g., memory protection is set after a power reset and then locked to prevent modification. This is done using a lock-bit signal. If the system allows debugging operations and the lock-bit can be overridden in a debug mode, the system configuration controls are not properly protected.

1271: Uninitialized Value on Reset for Registers Holding Security Settings. Security-critical information stored in registers should have a known value when being brought out of reset. If that is not the case, these registers may have unknown values that put the system in a vulnerable state.

1280: Access Control Check Implemented After Asset is Accessed. Access control checks are required in hardware before security-sensitive assets like keys are accessed. If this check is implemented after the access, then the check is clearly useless.

1276: Hardware Child Block Incorrectly Connected to Parent System. Hardware blocks are connected to a parent system that controls their inputs. If an input is incorrectly connected, affecting security attributes like resets while maintaining correct functionality; the integrity of the data of the child block can be violated.

1245: Improper Finite State Machines (FSMs) in Hardware Logic. FSMs are used in hardware to carry out different functionality according to different states. When FSMs are used in modules that control the level of security a system is in, it becomes important that the FSM does not have any undefined states. These undefined states may allow an adversary to carry out functionality that requires higher privileges. An improper FSM can present itself as unreachable states, FSM deadlock, or missing states.

2.5 Prompt Engineering

Prompt engineering is crucial to the performance of an LLM. Careful prompt engineering outperforms the baseline LLM performances in natural language tasks [44, 54]. A study exploring the use of Copilot [22] to solve CS1 level coding assignments has shown that tweaks to the prompt improve the performance from around 50% to 60% [17]. Prompt variations are also important in improving the results of text-to-image generation tasks[29, 36]. Thus prompt engineering is crucial when using LLMs for code repair.

3 DESIGNS AND BUGS

To explore the idea of using LLMs to fix HW security bugs, we first collate and prepare a set of benchmark designs, coming up with ten hardware security bugs from three sources. The sources are CWE descriptions on the MITRE website [15], OpenTitan System-on-Chip (SoC) [1] and the Hack@DAC 2021 SoC [24]. Each bug is represented in a design, as described in Table 2.

3.1 MITRE's CWEs

We use examples provided in MITRE's hardware design list to come up with simple designs that may represent CWE(s). The bugs and corresponding fixes for this source are shown in Figure 2.

3.1.1 *Locked Register.*

This design has a register that is protected by a lock bit. The contents of the register may only be changed when the `lock_status` bit is low. In Figure 2(a), a `debug_unlocked` signal overrides the `lock_status` signal allowing the locked register to be written into even if `lock_status` is asserted.

3.1.2 *Lock on Reset.*

This design has a register that holds sensitive information. This register should be assigned a known value on reset. In Figure 2(b), the register `locked` should have a value assigned under reset, but in this case, there is no reset block.

3.1.3 *Grant Access.*

This design contains a register that should only be modifiable if the `usr_id` input is correct. In Figure 2(c), the register `data_out` is assigned a new value if the `grant_access` signal is asserted. This should happen when `usr_id` is correct, but since the check happens after writing into `data_out` in blocking assignments, `data_out` may be modified when the `usr_id` is incorrect.

3.1.4 *Trustzone Peripheral.*

This design contains a peripheral instantiated in an SoC. To distinguish between trusted and untrusted entities, a signal is used to assign the security level of the peripheral. This is also described as a privilege bit used in Arm TrustZone to define the security level of all connected IPs. In Figure 2(d), the security level of the instantiated peripheral is grounded to zero, which could lead to incorrect privilege escalation of all input data.

3.2 Google's OpenTitan

OpenTitan is an open-source project designed to provide a silicon root of trust. It contains implementations of security measures that make the SoC secure. We inject bugs by tweaking the RTL of these security measures in different modules. The bugs and their corresponding fixes for this source are shown in Figure 3.

3.2.1 *ROM Control.*

This design contains a module that acts as an interface between the ROM and the system bus. The ROM has scrambled contents, and the controller descrambles the content for memory requests. We target the `COMPARE.CTRL_FLOW.CONSISTENCY` security measure in the `rom_ctrl_compare` module. A part of this measure is that the `start_i` signal should only be asserted in the `Waiting` state, otherwise, an alert signal is asserted. In Figure 3(a), because of our induced bug, the alert signal is incorrectly asserted when `start_i` is high in any state other than `Waiting`.

3.2.2 *OTP Control.*

This is a one-time programmable memory controller that provides the programmability for the device's life cycle. It ensures that the correct life cycle transitions are implemented

https://t.me/learningnets

```
module locked_register ( input [15:0] Data_in,
input clk, resetn, write, lock_status, debug_unlocked,
output reg [15:0] Data_out );
always @(posedge clk or negedge resetn) begin
    if (~resetn) begin
        Data_out <= 16'h0000;
    end
    else if (write&(~lock_status|debug_unlocked)) begin
    else if (write&~lock_status) begin
        Data_out <= Data_in;
    end
    else if (~write) begin
        Data_out <= Data_out;
    end
end
endmodule
```

(a) Locked Register: Bug - debug signal overrides lock status signal. Fix- remove debug signal in condition.

```
module lock_on_reset (
input wire clk, resetn, unlock, d,
output reg locked );
always @(posedge clk or negedge resetn) begin
    if(unlock) locked <= d;
    else locked <= locked;
    if (~resetn) locked <= 0;
    else if(unlock) locked <= d;
    else locked <= locked;
end
endmodule
```

(b) Lock on reset: Bug- register locked is not assigned a value under a reset condition. Fix- locked register is assigned 0 at reset.

```
module
        user_grant_access(data_out, usr_id, data_in, clk, rst_n);
output reg [7:0] data_out;
input wire [2:0] usr_id;
input wire [7:0] data_in;
input wire clk, rst_n;
reg grant_access;
always @ (posedge clk or negedge rst_n)
begin
    if (!rst_n) data_out = 0;
    else begin
        data_out = (grant_access) ? data_in : data_out;
        grant_access = (usr_id == 3'h4) ? 1'b1 : 1'b0;
        grant_access = (usr_id == 3'h4) ? 1'b1 : 1'b0;
        data_out = (grant_access) ? data_in : data_out;
    end
end
endmodule
```

(c) Grant access: Bug- `grant_access` signal is used before it is assigned a value. Fix- `grant_access` signal is used after it is assigned a value.

```
module soc(clk, rst_n, rdata, rdata_security_level, data_out);
input clk, rst_n, rdata_security_level;
input [31:0] rdata;
output [31:0]  data_out;
    tz_peripheral u_tz_peripheral(
    .clk(clk), .rst_n(rst_n), .data_in(rdata),
    .data_in_security_level(1'b0),
    .data_in_security_level(rdata_security_level),
    .data_out(data_out)  );
endmodule
```

(d) TZ peripheral: Bug- security level to peripheral is incorrectly grounded. Fix- security level for data is correctly assigned to parent signal.

Figure 2: MITRE CWE bugs and their corresponding repairs. The repair (green) replaces the bug (red) for a successful fix.

as the entity of the SoC changes among the 4 – Silicon Creator, Silicon Owner, Application Provider, and the End User. We target the `LCI.FSM.LOCAL_ESC` security measure in the `otp_ctrl_lci` module. A part of this measure is that the FSM jumps to an error state if the escalation signal is asserted. In Figure 3(b), no error is raised in such a case because of our induced bug.

```
// start_i
        should only be signalled when we're in the Waiting state
// SEC_CM: COMPARE.CTRL_FLOW.CONSISTENCY
logic start_alert;
assign start_alert = start_i && (state_q != Done);
assign start_alert = start_i && (state_q != Waiting);
```

(a) ROM Control: Bug- alert asserted when start is high in any state other than Done. Fix- alert asserted when start is high in any state other than Waiting.

```
if (escalate_en_i != lc_ctrl_pkg::Off || cnt_err) begin
  state_d = ErrorSt;

  fsm_err_o = 1'b1;
  if (error_q == NoError) begin
    error_d = FsmStateError;
  end
end
```

(b) OTP Control: Bug- alert is not raised when escalation signal is high. Fix- fsm alert signal is asserted appropriately.

```
StTx: begin
    valid = 1'b1;
    strb = {IfBytes{1'b1}};
    // transaction accepted
    if (kmac_data_i.ready) begin
      cnt_en = 1'b1;
      kmac_done_vld = 1'b1;

      // second to last beat
      if (cnt == CntWidth'(1'b1)) begin
        state_d = StTxLast;
      end
end
```

(c) Keymanager KMAC: Bug- kmac done signal is prematurely asserted. Fix- do not assert done signal here.

Figure 3: OpenTitan bugs and their corresponding repairs. The repair (green) replaces the bug (red) for a successful fix.

3.2.3 Keymanager KMAC. This design carries out the Keccak Message Authentication Code (KMAC) and Secure Hashing Algorithm 3 (SHA3) functionality. It is responsible for checking the integrity of the incoming message with the signature produced from the same secret key. We target the `KMAC_IF_DONE.CTRL.CONSISTENCY` security measure in the `keymgr_kmac_if` module. A part of this measure is that the kmac done signal should not be asserted outside the accepted window, i.e., when the FSM is in the done state. In Figure 3(c), because of our induced bug, the kmac done signal is incorrectly asserted in the transmission state `StTx`.

3.3 Hack@DAC-21

Hack@DAC-21 examples are bugs in the hardware designs for Hack@DAC 2021 CTF competition. Hack@DAC is a hackathon for finding vulnerabilities at the RTL level for a reasonably complex System-on-Chip (SoC). The bugs and their corresponding fixes for this source are shown in Figure 4.

3.3.1 Csr regfile. This design contains a module that carries out changes in control and status registers according to the system's state. This includes changes in privilege levels, incoming interrupts, virtualization, and cache support. We consider the module's functionality pertaining to the stalling of the core in the case of receiving an interrupt and/or debug request. In Figure 4(a), the debug signal overrides interrupt signals.

3.3.2 DMA. This design contains the Direct Memory Access module common to all blocks. It uses the memory address as input and

https://t.me/learningnets

```
// Wait for Interrupt
always_comb begin : wfi_ctrl
    // wait for interrupt register
    wfi_d = wfi_q;
    if (|mip_q || debug_req_i || irq_i[1]) begin
    if (|mip_q || irq_i[1]) begin
        wfi_d = 1'b0;
    end else if (!
        debug_mode_q && csr_op_i == WFI && !ex_i.valid) begin
        wfi_d = 1'b1;
    end
end
```

(a) Csr regfile: Bug- debug signal overrides interrupt signals. Fix- remove debug signal in condition.

```
riscv::pmp_access_t pmp_access_type_reg, pmp_access_type_new
    ; // riscv::ACCESS_WRITE or riscv::ACCESS_READ
reg pmp_access_type_en;
reg pmp_access_type_en;
always @(posedge clk_i or negedge rst_ni) begin
    if (!rst_ni) begin
        pmp_access_type_en <= 0;
```

(b) DMA: Bug- pmp enable register is not assigned a value on reset. Fix- pmp enable register is assigned 0 on reset.

```
s15: begin
        Out_data_final <= Out_data;
        ct_valid_out <= 1'b1;
        state <= s0;
    end

    default: begin
            state <= s0;
    end
endcase
```

(c) AES2 Interface: Bug- Incomplete case statements. Fix- add default case.

Figure 4: Hack@DAC-21 bugs and their corresponding repairs. The repair (green) replaces the bug (red) for a successful fix.

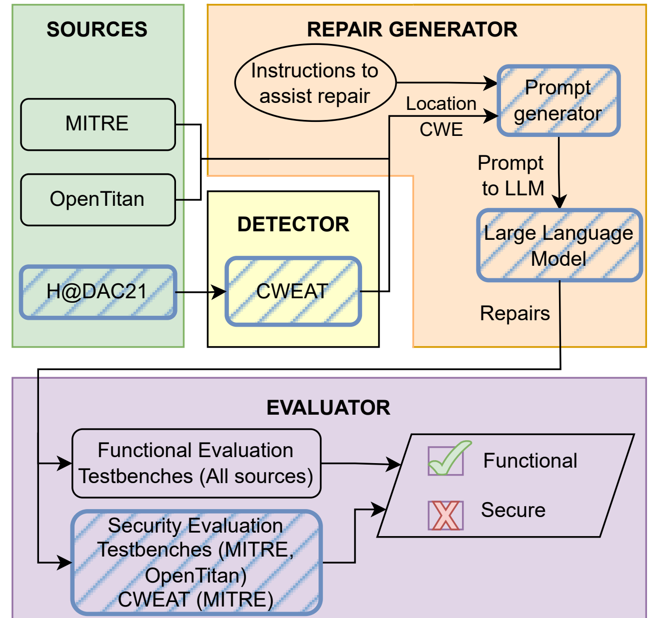

Figure 5: Overview of the framework used in our experiments It is broken down into 4 main components. Sources are the designs containing bugs. Detector localizes the bug (for bugs 8-10). Repair generator contains the LLM which generates the repairs. Evaluator verifies the success of the repair.

performs read or write operations according to the Physical Memory Protection (PMP) configuration. We consider the PMP access mechanism as the relevant security implementation. In Figure 4(b), the pmp register is not assigned any value on reset.

3.3.3 AES 2 Interface. This design instantiates the Advanced Encryption Standard (AES) module and outputs the cipher text to the system. It uses an FSM to interact with the AES (initialize, reset, and checking valid output). In Figure 4(c), the case statement has neither enough cases nor a default statement.

4 EXPERIMENTAL METHOD

To test the capability of LLMs to generate successful repairs, we design experiments that use the designs and bugs detailed in Section 3. In this section we present our framework that automates the execution of our experiments, starting from the identification of bugs to the evaluation of the repairs.

4.1 LLM-based Repair Evaluation Framework

The framework overview for our experiments is shown in Figure 5. It can be broken down into four components, i.e., the **Sources**, **Detector**, **Repair Generator**, and **Evaluator**. The Sources are discussed in Section 3, and the Detector, used for bugs from Hack@DAC-21, is discussed in Section 4.2.

4.1.1 Repair Generator. This block takes the **location** and **CWE** of the bug as the input from the Source or the Detector. For MITRE and OpenTitan, we assume that the location of the bugs, i.e., starting and ending line numbers and the filepath of the buggy file, is known. For Hack@DAC, we run a bug detector tool that gives us the location and relevant CWE of the bugs as its outputs.

For each bug, we develop **Instructions to assist repair**. These are comments before and after the buggy code that assist the LLMs in generating an appropriate repair for that bug. The **Prompt generator** combines the code before the bug, buggy code in comments, and instructions to form the **Prompt to LLM**. This can be worded as 'what the LLM sees'. An example of this construction is shown in Figure 6 (a)-(c) for the design Grant Access. The instructions are broken down into Bug Instruction and Fix Instruction. The former describes the nature of the bug and lets the LLM know that the bug follows. The latter follows the bug in comments and instructs the LLM on how to fix the bug. These instructions are varied in different degrees of detail according to the bug as discussed in Section 4.3.1. The **Large Language Model** takes the **Prompt to LLM** as input and outputs the **Repairs**. The repairs produced may be correct or incorrect. Some of the repairs generated using the prompt Figure 6(c) are shown in Figure 6 (d)-(f).

4.1.2 Evaluator. This block takes the **Repairs** generated by the LLM and verifies their correctness by evaluating their functionality and security. A repair is successful if it is both **functional** and **secure**. We use ModelSim simulator as a part of Xilinx Vivado 2022.2 to simulate the designs and custom testbenches.

https://t.me/learningnets

Table 2: **Bugs Overview. We assign a CWE to each bug and give a description of the design.**

Bug	Design	CWE	Source	Description
1	Locked Register	1234	MITRE	This register module supports a lock mode that blocks any writes after lock is set to 1. However, it also allows override of the lock protection when scan_mode or debug_unlocked modes are active.
2	Lock on Reset	1271	MITRE	This register module supports a lock mode that allows writes after unlock is set to 1. The locked register does not have a value assigned on reset and when the circuit is brought out of reset, the state will be unknown.
3	Grant Access	1280	MITRE	This module allows register contents to be modified only when correct user id is used. However, the asset is allowed to be modified even before the access control check is complete.
4	Trustzone Peripheral	1276	MITRE	This module instantiates a peripheral within a SoC using a signal to distinguish between trusted and untrusted entities. However, this signal depicting the security level is incorrectly grounded.
5	ROM Control	1245	OpenTitan	This module contains an FSM where an alert should be triggered if start signal is high in any state other than Waiting. However, the state is incorrectly compared to the Done state instead of the Waiting state.
6	OTP Control	1245	OpenTitan	The life cycle interface FSM should move into an invalid state upon global escalation via life cycle. However, the corresponding error signal for this transition is not asserted when the escalation signal is high.
7	Keymanager KMAC	1245	OpenTitan	This module has an FSM which has a done signal which should only be asserted at the time of completion. However, this signal is asserted outside of expected window, i.e., during a transmission state.
8	Csr regfile	1234	H@DAC-21	If there is any interrupt pending or an incoming interrupt request is received, the core should be unstalled. In this example, the core is also unstalled if there is a request to enter debug mode.
9	DMA	1271	H@DAC-21	This module has a security sensitive register that controls whether the PMP (Physical Memory Protection) register can be written into. This register should be assigned a value on reset but it is not.
10	AES-2 interface	1245	H@DAC-21	The FSM for AES 2 interface has a total of 15 states and does not include a default statement for its 4 bit state variable. This represents an incomplete case statement of an FSM.

Functional Evaluation is done using custom testbenches we developed in Verilog. These are made for each design and contain tests to check for various input vectors. A failed testbench indicates a failure of at least one test or a syntax error in the design. For MITRE designs, we develop testbenches that cover the design's entire intended functionality. For OpenTitan and Hack@DAC designs, we cover partial functionality for inputs and outputs that pertain to the buggy code. These designs require an additional step of forming the Device Under Test (DUT) before simulation. This entails tracking the files instantiated by the buggy file and the files that need to be analyzed before the buggy file. This list of files is input to the simulator.

Security Evaluation is done through a combination of custom testbenches (for MITRE and OpenTitan) and CWEAT (for Hack@DAC). For MITRE designs, the tests are designed according to the weaknesses mentioned on the MITRE website for each bug. For OpenTitan, we use the security countermeasures defined in their relevant .hjson files for the peripherals. It is difficult to verify the security countermeasure completely because that requires simulating the entire SoC through the software for Design Verification by OpenTitan. This method is still a work in progress for the OpenTitan team. The countermeasures that can currently be verified completely still require a lot of simulation time. Hence, we develop custom testbenches that verify very specific functionality for the bugs we introduce in the OpenTitan designs. For Hack@DAC, we employ CWEAT for security evaluation; this is discussed in Section 4.2.

Functional and Security Verification are not always mutually exclusive. There is often an overlap between the two, e.g., for CWE 1271 bugs, the security verification requires both a value on reset for the security-sensitive register and the correct lock mechanism. The latter is also a requirement for correct functionality. In the case of Bug 3, the functional verification is a subset of the security evaluation because the goal of the design is to grant user access under the correct input.

4.2 End-to-end example with CWEAT

We present a demonstrative end-to-end framework for the detection and repair of some CWEs in Verilog. This includes the detection of the bug, the generation of repair using this detection, and the evaluation of the correctness of the repair generated. The elements of this pipeline are represented in hatched blocks in Figure 5.

The **Detector** used is a static analysis tool that has the capability to detect some weaknesses at the RTL. We use the methods described in [5] to traverse the Abstract Syntax Trees (AST)s generated by the Verific Verilog parser. There is one AST produced per module. Each node of the tree represents a syntactical element of the RTL code with various information about identifiers, types, values and conditions. The ASTs are traversed using keywords and patterns to indicate potential vulnerabilities in CWEs 1234, 1271, and 1245. We ran this tool over the Hack@DAC 2021 SoC and selected three instances, one per CWE, for the purposes of this paper. We use the same tool for **security evaluation** of the generated responses. We replace the buggy code with the repaired code in the SoC and run the tool again. If the same bug is picked up, i.e., the same location and CWE, we can determine that the repair is not successful. If that is not the case, we infer that the repair is adequate.

We envision the use of this (or similar) LLM-infused end-to-end solution by RTL designers as they write Hardware Description Language (HDL) code in the early stages of Hardware Design. CWEAT can highlight the potential weakness to the designer, run it through the LLM to produce repairs, choose the ones that are secure, and present those as suggestions to the designer.

4.3 Experimental Parameters

LLMs have several parameters that can be manipulated to produce responses. We change the prompt (as discussed in Section 4.1.1) according to the bug and instructions. We also vary the **Instructions**, **Temperature** and **Models** while keeping the **top_p**, **number_of_completions (n)** and **max_tokens** constant at 1, 20 and

https://t.me/learningnets

```verilog
module user_grant_access(data_out, usr_id,data_in,clk,rst_n);
output reg [7:0] data_out;
input wire [2:0] usr_id;
input wire [7:0] data_in;
input wire clk, rst_n;
reg grant_access;
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) data_out = 0;
    else begin
        data_out = (grant_access) ? data_in : data_out;
        grant_access = (usr_id == 3'h4) ? 1'b1 : 1'b0;
    end
end
endmodule
```

(a) Original buggy file for bug 3 in design Grant Access.

```verilog
// BUG: Access Control Check Implemented after
// asset is Accessed.

// Ensure that access is granted before data is
// accessed.
// FIX:
```

(b) Instructions used to assist repair utilizing variation c. The Bug instruction is highlighted in yellow and the Fix instruction is highlighted in gray.

```verilog
module user_grant_access(data_out,usr_id,data_in,clk,rst_n);
output reg [7:0] data_out;
input wire [2:0] usr_id;
input wire [7:0] data_in;
input wire clk, rst_n;
reg grant_access;
always @(posedge clk or negedge rst_n) begin
    if (!rst_n)
        data_out = 0;
    else begin
// BUG: Access Control Check Implemented after
// asset is Accessed.
// data_out = (grant_access) ? data_in : data_out;
// grant_access = (usr_id == 3'h4) ? 1'b1 : 1'b0;
// Ensure that access is granted before data is
// accessed.
// FIX:
```

(c) Prompt-to-LLM: What the LLM gets as an input.

```verilog
grant_access = (usr_id == 3'h4) ? 1'b1 : 1'b0;
if (grant_access) data_out = data_in;
```

(d) Correct repair produced by `code-davinci-001` at temp=0.7.

```verilog
data_out = (grant_access) ? data_in : data_out;
grant_access = (usr_id == 3'h4) ? 1'b1 : 1'b0;
```

(e) Incorrect repair produced by `code-davinci-001` at temp=0.5. This response passed the functional evaluation but failed the security evaluation.

```verilog
if (grant_access) data_out = data_in;
```

(f) Incorrect repair produced by `code-davinci-001` at temp=0.7. This response failed the functional and security evaluations.

```verilog
module user_grant_access(data_out,usr_id,data_in,clk,rst_n);
output reg [7:0] data_out;
input wire [2:0] usr_id;
input wire [7:0] data_in;
input wire clk, rst_n;
reg grant_access;
always @(posedge clk or negedge rst_n) begin
    if (!rst_n)
        data_out = 0;
    else begin
        grant_access = (usr_id == 3'h4) ? 1'b1 : 1'b0;
        if (grant_access) data_out = data_in;
    end
end
endmodule
```

(g) Generated repair file using 6(d). This is sent to the Evaluator for evaluation.

Figure 6: Prompt to LLM and sample repairs produced for Bug 3 - Grant Access. Sub-figures (a)-(c) show how the bug is combined with instructions to generate the prompt that the LLM gets as one of its inputs. Sub-figures (d)-(f) show some actual repairs generated by an LLM.

Table 3: Instruction Variations. We develop 5 types to assist repair of bugs. Variation a is the base variation with no assistance. The level of detail/assistance increases from variation a to e.

Instruction Variation	Description
a	No Instruction
b	Natural language description of bug
c	Natural language description of bug Prescriptive instruction of how to fix
d	Natural language description of bug Descriptive instruction of how to fix
e	Code examples of bug and fix

200 respectively. **top_p** is an alternative to sampling with temperature, called nucleus sampling, where only results with probability mass of **top_p** are considered. **n** is the number of completions generated by the LLM per request. **max_tokens** is the maximum number of tokens that can be generated per completion.

4.3.1 Instruction Variation . We test five instruction variants to guide the repair of bugs. They are described in Table 3. Each variation has 2 parts – **Bug Instruction** and **Fix Instruction**. The former describes the nature of the bug and precedes the commented bug. The latter follows the bug in comments and represents guidance to the LLM on how to fix the bug.

Variation **a** provides no assistance and is the same across all bugs. The **Bug instruction** is **"`BUG:`"** and the **Fix Instruction** is **"`FIX:`"**. The **Bug Instruction** for the remaining variations is a description of the nature of the bug. We take inspiration from the MITRE website and cater them according to the CWE they represent. For variation **e** this description is appended with an example of a 'generalized' bug in comments and its fix without comments. This generalization is done through using more common signal names and coding patterns. The **Fix Instruction** for **b** and **e** is the same as that for **a**. For **c**, it is preceded by a 'prescriptive' instruction which means that natural language is used to assist the fix. For **d**, however, it is preceded by a 'descriptive' instruction which means that language resembling pseudo-code is used to assist the fix. The components of instruction that change are shown in Table 4.

4.3.2 Temperature (t). A higher value means that the LLM takes more risks and yields more creative completions. We use $t \in \{0.1, 0.3.0.5, 0.7, 0.9\}$.

4.3.3 Models. We use four LLMs, three of which are made available by OpenAI [35] and one is an open-source model available through [34]. The OpenAI Codex models are derived from GPT-3 and were trained on millions of public GitHub repositories. They can ingest and generate code, and also translate natural language to code. We use/evaluate `code-davinci-001`, `code-davinci-002` and `code-cushman-001` models. From Hugging Face, we evaluate the model `CodeGen-16B-multi`, which we refer to as `CodeGen` in this work. It is an autoregressive language model for program synthesis trained sequentially on The Pile and BigQuery.

https://t.me/learningnets

Table 4: Details of Instruction Variations and Stop keywords used. The same Bug instruction is used for variations b,c,d, shown in column 2. In case of variation e, this Bug instruction (in column 2) is appended by an example of a bug and its repair in comments, shown in column 3. Fix instructions for variations c and d precede the string "FIX:", shown in columns 4 and 5 respectively. Additional stop keywords that terminate the further generation of tokens by LLMs are shown in column 6.

Bug	Bug Instruction for variations b,c,d,e	Bug Instruction appended for variation e	Fix Instruction for variation c	Fix Instruction for variation d	Stop keywords

1	// BUG: Hardware Internal or Debug Modes Allow Override of Locks.	// if (write & (~lock | debug)
		// FIX:
		if (write & ~lock)
			// Debug or scan signals should not be included in if condition	// Write data if write signal high and lock signal is low	'\n'

2	// BUG: Uninitialized Value on Reset for Registers Holding Security Settings.	// if(unlock) q <= d_in; else q <= q;
		// FIX:
		if(~resetn) q<=0;
		else if(unlock) q <= d_in; else q <= q;
			// Ensure that the security sensitive lock register is assigned a value on reset.	// Assign 0 to register when reset is low	-

3	// BUG: Access Control Check Implemented After Asset is Accessed.	// d = (access) ? d_in : d_out;
		// access = (id == 2'h2) ? 1'b1 : 1'b0;
		// FIX:
		access = (id == 2'h2) ? 1'b1 : 1'b0;
		d = (access) ? d_in : d_out;
			// Ensure that access is granted before data is accessed.	// Assert access when id is correct. Then assign data to register if access is asserted.	end

4	// BUG: Hardware Child Block Incorrectly Connected to Parent System	// .in_security_level(1'b0), // FIX: .in_security_level(data_security_level),	// The security level of the child signal should match that of the parent signal	// assign data security level to input security level	'\n'

5	// BUG: Incorrect Alert Mechanism	// alert = start && (state!=FINISHED);
		// FIX:
		alert = start && (state!=IDLE);
			// An alert signal should be set if an FSM is instructed to start in a state that is not idle	// Assert alert signal if start signal is asserted and state is not idle	'\n'

6	// BUG: Escalation does not lead to fatal error	// if (escalate_i != 0) begin
		// state_d = err_state;
		// FIX:
		if (escalate_i != 0) begin
		state_d = err_state; fsm_err_o = 1'b1;
			// FSM should raise error if system is in escalation	// Assert error when escalation input is high	end

7	// BUG: Done signal is asserted prematurely	// if (ready) begin done_vld = 1'b1;
		// FIX:
		if (ready) begin done_vld = 1'b0;
			// Do not assert done signal in intermediate states	// assign zero to done signal in ready state	end

8	// BUG: Hardware Internal or Debug Modes Allow Override of Locks.	// BUG: Hardware Internal or Debug Modes Allow Override of Locks.	// Debug or scan signals should not be included in if condition	// unstall core when interrupt is high	'\n'

9	// BUG: Uninitialized Value on Reset for Registers Holding Security Settings.	// if(unlock) q <= d_in; else q <= q;
		// FIX:
		if(~resetn) q<=0;
		else if(unlock) q <= d_in; else q <= q;
			// Ensure that the security sensitive lock register is assigned a value on reset.	// Assign 0 to register when reset is low	-

10	// BUG: Incomplete case statement	// endcase
		// FIX:
		default: begin
		state <= s0; end endcase
			// Add a default case statement	// Write a default case statement where initial state is assigned to state	endcase

4.3.4 Number of lines before bug. Another parameter to consider is in the prompt preparation: the number of lines of existing code given to the LLM. Some files may be too large for the entire code before the bug to be sent to the LLM. We, therefore, select a minimum of 25 and a maximum of 50 lines of code before the bug as part of the prompt. In Figure 6(a), this would be lines 1–9 (inclusive). If there are more than 25 lines above the bug, we include enough lines that go up to the beginning of the block the bug is in. This block could be an always block, module, or case statement, etc.

4.3.5 Stop keywords. A stop keyword is specified to stop the response of the LLM (i.e., the response is considered finished when the stop keyword is generated by the model). They are not included in the response. We developed a strategy that works well with the set of bugs we have. The default stop keyword is `endmodule`. Additional keywords used are shown in the column Stop keywords in Table 4.

https://t.me/learningnets

5 EXPERIMENTAL RESULTS

We set up our experimental framework for each LLM, generating 20 responses for every combination of bug, temperature, and instruction variation. The responses are counted as successful repairs if they pass functional and security tests. The number of successful repairs is shown as heat-maps in Figure 7. The maximum value for each element is 20, i.e., when all responses were successful repairs.

5.1 RQ1: Can LLMs fix hardware security bugs?

This work shows that LLMs can repair simple hardware bugs. `code-davinci-001`, `code-davinci-002`, and `code-cushman-001` yielded at least one successful repair for every bug in our dataset. `CodeGen` was successful for 7 out of 10 bugs. In total, we requested 20,000 repairs out of which 6376 were correct, a success rate of 31.9%. The key here lies in selecting the best-observed parameters for each LLM. `code-davinci-001` performs best at variation *d*, *temp* 0.1 producing 71% correct repairs. `code-davinci-002`, `code-cushman-001` and `CodeGen` perform best at $(e, 0.1)$, $(d, 0.1)$ and (a, 0.3 and 0.5) with success rates of 70%, 58% and 12% respectively. Performance of these LLMs across bugs is shown in Figure 8.

We can fine-tune the parameters for each bug. The choice of the right combination of model, instruction variations and temperature can yield near-perfect results. We present these best-observed settings in Table 5. Under these settings, Bug 7 has a success rate of 85% and the rest have a success rate of 100%.

Table 5: Best-observed settings for each bug. 'dv1', 'dv2' and 'cus' stand for `code-davinci-001`, `code-davinci-002` and `code-cushman-001`. Within the settings arrays, the first element is the LLM, the second is a set of instruction variations and the third is a set of temperatures.

Bug	Best Setting
1	[dv1,b,0.1] [dv2,(b,d),(0.1,0.3)] [dv2,c,(0.1,0.3,0.5)]
2	[cus,d,0.1] [dv1,e,0.1] [dv2,e,(0.1,0.3)]
3	[cus,a,0.1] [dv1,c,(0.1,0.3)] [dv1,d,0.1] [dv2,(b,c,e),(0.1,0.3,0.5)] [dv2,d,(0.1,0.3,0.5,0.7)]
4	[cus,(a,c,d),(0.1,0.3,0.5)] [cus,b,(0.1,0.3,0.5,0.7)] [dv1,(a,b),(0.1,0.3,0.5)] [dv1,(c,d),(0.1,0.3,0.5,0.7)] [dv2,(a,b,d),(0.1,0.3,0.5,0.7)] [dv2,c,(0.1,0.3,0.5,0.7,0.9)] [dv2,e,0.1]
5	[cus,(c,e),0.1] [dv2, (a,c),(0.1,0.3,0.5)] [dv2,b,0.1] [dv2, (d,e),(0.1,0.3)]
6	[cus,e,(0.1,0.3)]
7	[dv1,d,0.1]
8	[dv1,(b,e),0.1] [dv1,c,(0.1,0.3,0.5)] [dv2,c,(0.1,0.3,0.5)] [dv2,e,(0.1,0.3)]
9	[cus,e,(0.1,0.3,0.5)] [dv1,e,(0.1,0.3)] [dv2,e,(0.1,0.3,0.7)]
10	[cus,(c,d),0.1] [dv1,a,0.1] [dv1,(b,d),(0.1,0.3)] [dv1,c,(0.1,0.3,0.5)] [dv2,(c,d),0.1]

5.2 RQ2: What bugs are amenable to repair?

The cumulative number of correct repairs for each bug is shown in Figure 9. Bugs 3 and 4 were the best candidates for repair with success rates of over 50%. These are examples from MITRE where the signal names used clearly indicate their intended purposes. For the **Grant Access** module, the signals of concern are `grant_access` and `usr_id` used in successive lines. LLMs are able to interpret the intended functionality that the `usr_id` should be compared before granting access. Most successful repairs either flipped the order of blocking assignments or lumped them into an assignment using the ternary operator. Similarly, **Trustzone Peripheral** uses signal names `data_in_security_level` and `rdata_security_level` which illustrate their intended functionality.

Bugs 2, 6, 7, and 9 were the hardest to repair with success rates of under 25%. Bugs 2 and 9 had the same bug of a register holding security settings not initialized under reset. This was difficult to repair because a fix required the creation of an always block with an appropriate reset as well as re-creating the previous intended functionality. Bug 7 was the hardest to repair because it was the only bug that required a line to be removed without replacement as a fix. We hypothesize that Bug 6 was hard to fix because it was difficult to phrase the fix instruction according to the description of the bug provided by Opentitan. The fix relies on asserting the fsm alert signal when escalation signal is high, but this condition is represented in code as `if (escalate_en_i != lc_ctrl_pkg::Off)` which is harder to grasp by the LLMs.

5.3 RQ3: How should prompts be engineered to assist the repair of hardware bugs?

The 5 variations from *a* to *e* increase in the level of detail. In general, apart from CodeGen, all LLMs do better with more detail being provided to assist repair as shown in Figure 10(a). Variations *c*-*e* perform better than variations *a* and *b*. They include a fix instruction after the buggy code in comments, giving credence to the use of two separate instructions per prompt (one before and one after the bug in comments). Variation *d* has the highest success rate among OpenAI LLMs and is therefore our recommendation for bug fixes. The use of a fix instruction in 'pseudo-code' fashion leads to the best results. There is variation within LLMs for the best-observed instruction variation, e.g., `code-davinci-002` and `CodeGen` perform best at *e*.

5.4 RQ4: Does the temperature matter?

A higher temperature allows the LLM to be more creative in its responses. As shown in Figure 10(b), the LLMs perform better at lower temperatures. All OpenAI LLMs perform best at $t = 0.1$ and CodeGen performs best at 0.3. A lower temperature leads to less variation in responses as well, implying that the less creative responses are more likely to be correct repairs.

5.5 RQ5: Are some LLMs better than others?

The `code-davinci-002` LLM was the best performing, producing 2371 correct repairs out of 5000, giving it a success rate of 47.4%. `code-davinci-001`, `code-cushman-001` and `CodeGen` had success rates of 40.4%, 33.1% and 6.68% respectively. The large difference between OpenAI LLMss and `CodeGen` is caused by `CodeGen`

https://t.me/learningnets

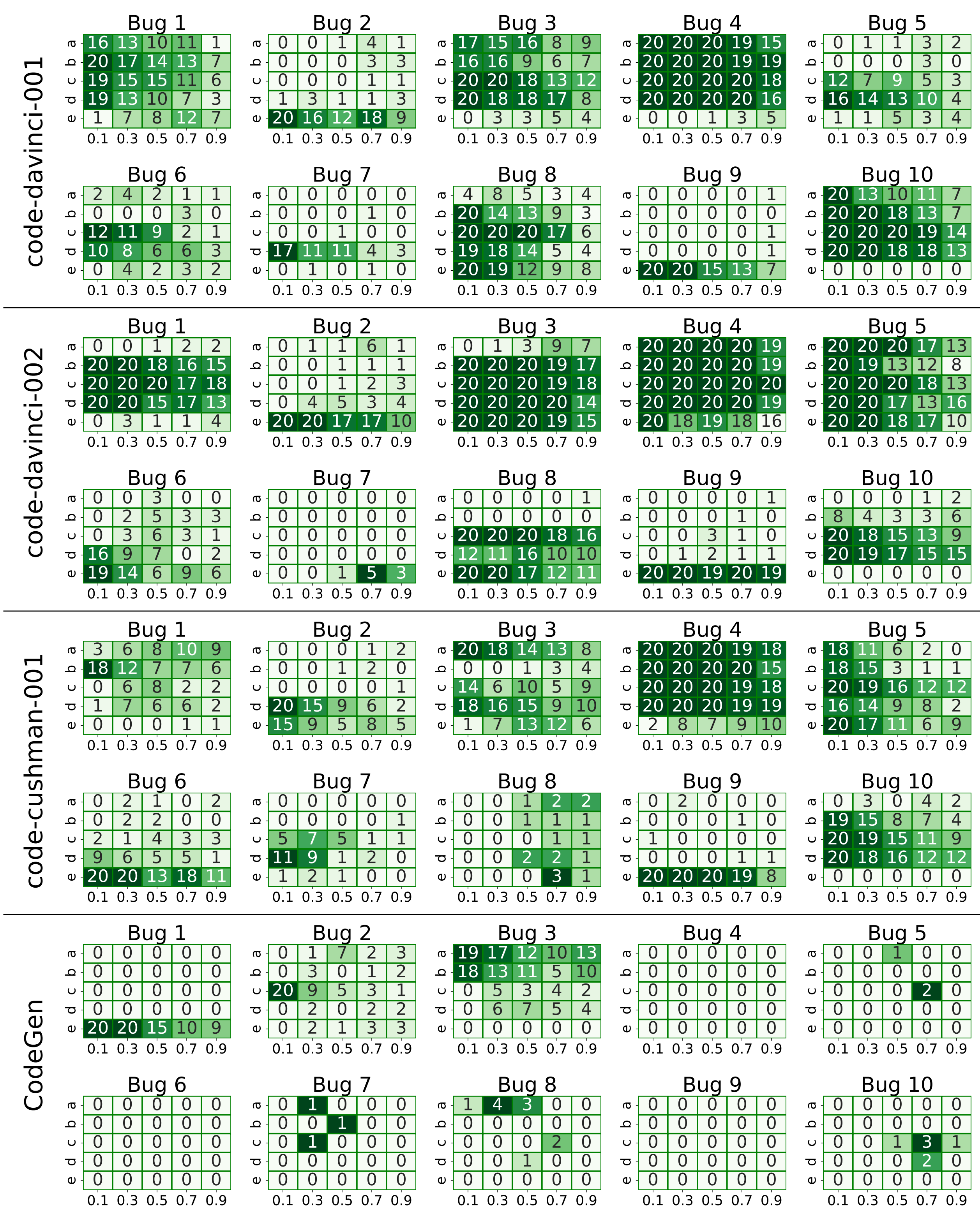

Figure 7: Results for all 4 LLMs represented as heatmaps. The maximum value for each small box is 20. A higher value indicates more success by LLM in generating repair and is highlighted with a darker shade. All bugs were repaired at least once by at least one LLM.

https://t.me/learningnets

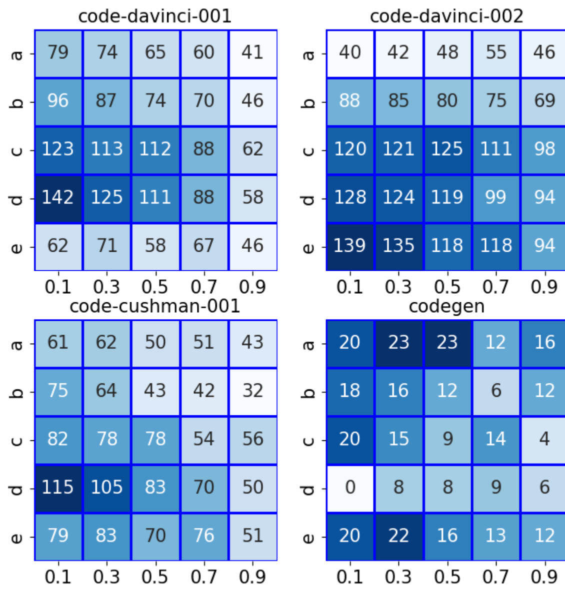

Figure 8: Results showing the performance of each LLM across all bugs in the form of heatmaps. Each small square shows the number of correct repairs for the corresponding instruction variation and temperature of the LLM. The maximum possible value is 200. A higher value indicates more success in generating repairs and is shaded in a darker color.

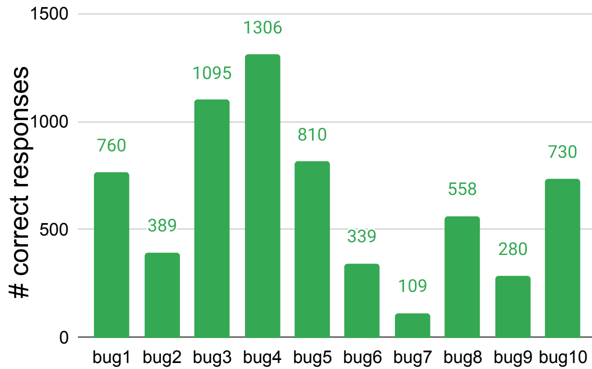

Figure 9: Number of correct repairs per bug. The number above each bar shows the sum of successful repairs across all LLMs for the corresponding bug. The maximum possible value is 2000. A higher value indicates that the bug was repaired more times.

being a much smaller LLM, having 16 billion parameters compared to the OpenAI LLMs that are based on GPT-3's ~175B parameters (the exact number of parameters for each of the OpenAI LLMs are not public). Additionally, `code-cushman-001` is slightly inferior to the `davinci` LLMs because it was designed to be quicker. This may mean that it has fewer parameters or that it has been trained over less data or both.

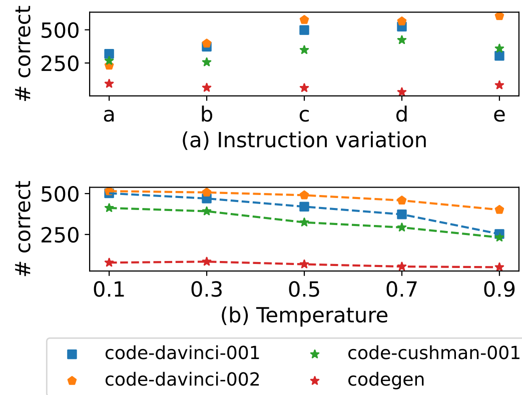

Figure 10: Results: Trends Across Models. The top graph shows the number of correct repairs for LLMs for specified instruction variations. The bottom graph shows the number of correct repairs for LLMs for specified temperature. The maximum value for each data point is 1000.

5.6 Comparison with CirFix

A comparison of our work with CirFix is shown in Table 6. We use the best-performing LLM (`code-davinci-002`) at $t = 0.1$ and generate one repair each for variations *a* and *b*. This is done to closely mirror the use case of CirFix. By comparing the first example produced by the LLM, we evaluate only one attempt at repair. This attempt is manually evaluated for correctness. We use variation a for the primary comparison as this variation uses no instructions to assist repairs. This variation produces 17.5 correct repairs as compared to CirFix's 16. The half repair corresponds to fixing one numeric error out of 2 for the first benchmark. To elicit the power of LLMs, we use variation b which includes a description of the type of bug to assist repair. We use the brief descriptions of bugs provided in CirFix's GitHub repository. Variation *b* fixes 20 of the 32 benchmarks and collectively, LLMs (both variations) are able to repair 23.5 of the bugs.

6 DISCUSSION AND LIMITATIONS

Our results show that LLMs have a lot of potential for bug repair. At the present, some assistance is required from the designer to identify the location and nature of the bug. This may be overcome by using other tools to localize the bugs and better design practices such as comments explaining the functionality of the design. Currently, the designer may also be needed to pick between a few options produced by the LLMs. This is where static analysis tools like CWEAT and other bug detection tools may come in to complete the loop by suggesting a repair that is correct to a high degree of confidence.

A limitation of the work is the subjectivity of instruction variations. Although the Bug instructions devised are inspired by the descriptions in CWEs, the Fix instructions are devised according to the knowledge and experience of the authors. Our work reveals the importance of these variations as subtle changes can affect the response generated by LLMs. Devising 5 categories is an attempt

https://t.me/learningnets

Table 6: Comparison on CirFix benchmarks. A successful repair is shown as y. We use two instruction variations for this comparison. An element - | y means that the repair using variation *a* was not successful but using variation *b* was. The element 1/2 means that 2 errors were used in the description of a single fault/bug and 1 out of the 2 was successfully repaired.

<table>
<tr><th>Project</th><th>Defect Description</th><th>CirFix</th><th>LLM
var a | b</th></tr>
<tr><td>decoder
(3 to 8)</td><td>Two numeric errors</td><td>y</td><td>1/2 | 1/2</td></tr>
<tr><td></td><td>Incorrect assignment</td><td>-</td><td>- | -</td></tr>
<tr><td>first_counter
overflow</td><td>Incorrect sensitivity list</td><td>y</td><td>y | y</td></tr>
<tr><td></td><td>Incorrect increment of counter</td><td>y</td><td>- | y</td></tr>
<tr><td></td><td>Incorrect reset</td><td>y</td><td>y | y</td></tr>
<tr><td>flip_flop</td><td>Incorrect conditional</td><td>y</td><td>- | -</td></tr>
<tr><td></td><td>if-statement branches swapped</td><td>y</td><td>| y</td></tr>
<tr><td>fsm_full</td><td>Incorrect case statement</td><td>-</td><td>- | -</td></tr>
<tr><td></td><td>Assignment to next state and
default in case statement omitted</td><td>y</td><td>y | y</td></tr>
<tr><td></td><td>State omitted from senslist</td><td>-</td><td>- | y</td></tr>
<tr><td></td><td>Incorrect blocking assignments</td><td>-</td><td>- | -</td></tr>
<tr><td>lshift_reg</td><td>Incorrect blocking assignments</td><td>y</td><td>- | y</td></tr>
<tr><td></td><td>Incorrect conditional</td><td>y</td><td>- | y</td></tr>
<tr><td></td><td>Incorrect sensitivity list</td><td>y</td><td>y | y</td></tr>
<tr><td>mux_4_1</td><td>Three numeric errors</td><td>-</td><td>y | y</td></tr>
<tr><td></td><td>Hex instead of binary numbers</td><td>-</td><td>y | y</td></tr>
<tr><td></td><td>1 bit instead of 4 bit output</td><td>-</td><td>y | y</td></tr>
<tr><td>i2c</td><td>Incorrect sensitivity list</td><td>y</td><td>- | -</td></tr>
<tr><td></td><td>Incorrect address assignment</td><td>-</td><td>y | -</td></tr>
<tr><td></td><td>No command acknowledgement</td><td>y</td><td>y | y</td></tr>
<tr><td>sha3</td><td>Off-by-one error in loop</td><td>y</td><td>y | -</td></tr>
<tr><td></td><td>Incorrect assignment to wire</td><td>-</td><td>y | -</td></tr>
<tr><td></td><td>Skipped buffer overflow check</td><td>y</td><td>- | -</td></tr>
<tr><td></td><td>Incorrect bitwise negation</td><td>-</td><td>y | y</td></tr>
<tr><td>tate_pairing</td><td>Incorrect logic for bitshifting</td><td>-</td><td>- | -</td></tr>
<tr><td></td><td>Incorrect instantiation of modules</td><td>-</td><td>y | y</td></tr>
<tr><td></td><td>Incorrect operator for bitshifting</td><td>-</td><td>y | y</td></tr>
<tr><td>reed_solomon
decoder</td><td>Insufficient register size
for decimal values</td><td>-</td><td>- | -</td></tr>
<tr><td></td><td>Incorrect sensitivity list for reset</td><td>y</td><td>y | y</td></tr>
<tr><td>sdram
controller</td><td>Incorrect assignments to registers
during synchronous reset</td><td>y</td><td>y | 1/2</td></tr>
<tr><td></td><td>Numeric error in definitions</td><td>-</td><td>y | y</td></tr>
<tr><td></td><td>Incorrect case statement</td><td>-</td><td>- | y</td></tr>
<tr><td></td><td></td><td>16</td><td>17.5 | 20</td></tr>
</table>

to standardize this process, but more varieties are probably needed to study their effects better. Moreover, instructions are challenging to generalize across different bugs. Ideally, a designer would want variation **a** to fix all bugs because no instructions are needed. But since more information is needed according to the particular instance of the bug for a higher probability of a successful fix, it is a challenge to form a small set of instructions, e.g., if LLMs are able to produce a successful fix with the Bug Instruction "Improper FSM" instead of "FSM has an unreachable state", that would be better.

Another limitation of the current framework is that the functional and security evaluations are not exhaustive. Security evaluation is dependent on the security objectives for the design and can not truly be exhaustive [4]. With this in mind, we limit the security evaluation to the particular bug that makes the design insecure. Ideally, efforts should be made to check that a fix does not result in another kind of bug. Functional evaluation is needed because a design that is secure but not functional is useless. For the CWE examples, we were able to build exhaustive testbenches because the designs were low in complexity and had only one or two modules. Ideally, the functional testbenches should be exhaustive for other examples too. But this would be very time-consuming as the size of the designs gets very large. It would be a difficult task to write testbenches for these complex SoCs and simulating the designs according to the software provided by OpenTitan and Hack@DAC takes a lot of time, e.g., design verification of OpenTitan examples takes ~10 minutes and it takes ~15 minutes to simulate the Hack@DAC-21 SoC. Therefore, we chose to build custom testbenches that test the code a repair could impact.

The use of end-tokens is another area of subjectivity that influences the success rate of repairs. Some strategies are intuitive like using the end line token as an end token for a bug that is present in only one line. Others may require more creativity because some lines of code can be written in multiple ways. A repair that spans multiple conditional statements, e.g.,

```
if (~resetn) begin locked <= 0; end
else if(unlock) begin locked <= d; end
else begin locked <= locked; end
```

may not be completely produced if the keyword **end** is used as a stop token. On the other hand, not limiting a response with an appropriate stop token may mean that the LLM produces the correct repair but then adds more code that contains a syntax error or affects functionality. We use a post-processing script to minimize syntax errors. This involves adding/removing the `end` keyword as needed. When the LLM generates a repair, that repair is a substitute for the bug only. The number of `begin` and `end` keywords are counted. If the numbers are same, nothing is to be done, and the repair is inserted in place of the bug. If the number of begins are greater by an amount n, `end` is added at the end of the repair n times. If the number of ends are greater by an amount n, the first n instances of `end` are removed.

The LLMs are very quick in generating repairs. The 20 responses per request are generated in under a minute. While trying to find a repair for a bug, a Verilog designer should have enough suggested repairs very quickly. The designer can then choose the best suggestion as the repair. In our experiments, we faced some challenge because of token limits set by the OpenAI API. Since we were generating thousands of requests with a limited number of token keys, we had to wait for a minute ever time we reached the limit. This raised our generation of repair time to ~20 minutes per LLM.

To evaluate security-related hardware bugs, a large number of benchmarks are needed that show these defects in designs. Our work takes a step in this direction. We believe more examples are needed to make more conclusive claims about repair techniques.

7 CONCLUSIONS AND FUTURE WORK

By choosing the right parameters and providing the right prompts, LLMs can fix the hardware bugs in our corpus. All the bugs had at least one successful repair and 9 of the 10 had 100% correct responses with the best set of parameters. We have found that in instances where signal names and comments implicate the functionality, LLMs have a high success rate. Conversely, fixes that span more

https://t.me/learningnets

than 1 line or require the removal of a buggy line are harder to repair. Detailed instructions to assist repair tend to achieve higher success rates with variation *d* using a Fix instruction that uses pseudo-code-like language performing the best. LLMs at lower temperatures and bigger LLMs perform better than LLMs at higher temperatures and LLMs with fewer parameters. LLMs do a better job at fixing function-related bugs in Verilog relative to the program repair mechanism in CirFix. We propose the following directions for future work:

- Test a hybrid approach for security-related bugs. Use Linters, Formal Verification, fuzzing, fault localization, and static analysis tools for detection and LLMs, oracle-guided modifying algorithms for repair. An ensemble of these options is likely to have more success than one technique alone.
- Fine-tune LLMs over HDLs and see if the performance improves. This improves the generation of functional code [45].
- Explore the repair of functional bugs using LLMs with the full sweep of parameters. We only used one set of parameters that performed the best in our experiments.
- Build a database of security-related hardware bugs. Ongoing efforts like Trusthub's Vulnerability Database [3] can be consolidated with our examples to build standard benchmarks.

REFERENCES

[1] 2019. Hardware | OpenTitan Documentation. https://docs.opentitan.org/hw/

[2] 2022. VC Formal. https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html

[3] 2023. Trust-Hub.org. https://trust-hub.org/#/vulnerability-db/soc-vulnerabilities

[4] Sohrab Aftabjahani, Ryan Kastner, Mark Tehranipoor, Farimah Farahmandi, Jason Oberg, Anders Nordstrom, Nicole Fern, and Alric Althoff. 2021. Special Session: CAD for Hardware Security - Automation is Key to Adoption of Solutions. In *2021 IEEE 39th VLSI Test Symposium (VTS)*. 1–10. https://doi.org/10.1109/VTS50974.2021.9441032 ISSN: 2375-1053.

[5] Baleegh Ahmad, Wei-Kai Liu, Luca Collini, Hammond Pearce, Jason M. Fung, Jonathan Valamehr, Mohammad Bidmeshki, Piotr Sapiecha, Steve Brown, Krishnendu Chakrabarty, Ramesh Karri, and Benjamin Tan. 2022. Don't CWEAT It: Toward CWE Analysis Techniques in Early Stages of Hardware Design. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design (ICCAD '22)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/3508352.3549369

[6] Hammad Ahmad, Yu Huang, and Westley Weimer. 2022. CirFix: automatically repairing defects in hardware design code. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 990–1003. https://doi.org/10.1145/3503222.3507763

[7] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. 2017. Register transfer level information flow tracking for provably secure hardware design. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 1691–1696. https://doi.org/10.23919/DATE.2017.7927266 ISSN: 1558-1101.

[8] Armaiti Ardeshiricham, Yoshiki Takashima, Sicun Gao, and Ryan Kastner. 2019. VeriSketch: Synthesizing Secure Hardware Designs with Timing-Sensitive Information Flow Properties. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1623–1638. https://doi.org/10.1145/3319535.3354246

[9] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 1–27. https://doi.org/10.1145/3360585

[10] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. TFix: Learning to Fix Coding Errors with a Text-to-Text Transformer. In *Proceedings of the 38th International Conference on Machine Learning*. PMLR, 780–791. https://proceedings.mlr.press/v139/berabi21a.html ISSN: 2640-3498.

[11] Mohammad Mahdi Bidmeshki, Yunjie Zhang, Monir Zaman, Liwei Zhou, and Yiorgos Makris. 2021. Hunting Security Bugs in SoC Designs: Lessons Learned. *IEEE Design & Test* 38, 1 (Feb. 2021), 22–29. https://doi.org/10.1109/MDAT.2020.3013727 Conference Name: IEEE Design & Test.

[12] Cadence. 2022. Jasper RTL Apps | Cadence. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. https://doi.org/10.48550/arXiv.2107.03374 arXiv:2107.03374 [cs].

[14] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2023. Neural Transfer Learning for Repairing Security Vulnerabilities in C Code. *IEEE Transactions on Software Engineering* 49, 1 (Jan. 2023), 147–165. https://doi.org/10.1109/TSE.2022.3147265 Conference Name: IEEE Transactions on Software Engineering.

[15] The MITRE Corporation. 2022. CWE - CWE-1194: Hardware Design (4.1). https://cwe.mitre.org/data/definitions/1194.html

[16] Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. 2015. Automated Synthesis of Optimized Circuits for Secure Computation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. Association for Computing Machinery, New York, NY, USA, 1504–1517. https://doi.org/10.1145/2810103.2813678

[17] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2022. Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language. https://doi.org/10.48550/arXiv.2210.15157 arXiv:2210.15157 [cs].

[18] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. 2019. Hardfails: Insights into Software-Exploitable Hardware Bugs. In *Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19)*. USENIX Association, Santa Clara, CA, USA, 213–230.

[19] Dawn Drain, Chen Wu, Alexey Svyatkovskiy, and Neel Sundaresan. 2021. Generating bug-fixes using pretrained transformers. In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming (MAPS 2021)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3460945.3464951

[20] Khashayar Etemadi, Nicolas Harrand, Simon Larsen, Haris Adzemovic, Henry Luong Phu, Ashutosh Verma, Fernanda Madeiral, Douglas Wikstrom, and Martin Monperrus. 2022. Sorald: Automatic Patch Suggestions for SonarQube Static Analysis Violations. https://doi.org/10.1109/TDSC.2022.3167316 arXiv:2103.12033 [cs].

[21] Mohammad Rahmani Fadiheh, Dominik Stoffel, Clark Barrett, Subhasish Mitra, and Wolfgang Kunz. 2019. Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 994–999. https://doi.org/10.23919/DATE.2019.8715004 ISSN: 1558-1101.

[22] GitHub. 2021. GitHub Copilot · Your AI pair programmer. https://copilot.github.com/

[23] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (Nov. 2019), 56–65. https://doi.org/10.1145/3318162

[24] HACK@EVENT. 2022. HACK@DAC21 – HacK@EVENT. https://hackatevent.org/hackdac21/

[25] jasperlint. 2022. Jasper Superlint App. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/jaspergold-superlint-app.html

[26] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1–19. https://doi.org/10.1109/SP.2019.00002 ISSN: 2375-1207.

[27] Xuan-Bach D. Le and Quang Loc Le. 2021. ReFixar: Multi-version Reasoning for Automated Repair of Regression Errors. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. 162–172. https://doi.org/10.1109/ISSRE52982.2021.00028 ISSN: 2332-6549.

[28] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[29] Vivian Liu and Lydia B Chilton. 2022. Design Guidelines for Prompt Engineering Text-to-Image Generative Models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems (CHI '22)*. Association for Computing Machinery, New York, NY, USA, 1–23. https://doi.org/10.1145/3491102.3501825

https://t.me/learningnets

[30] Yunlong Lu, Na Meng, and Wenxin Li. 2021. FAPR: Fast and Accurate Program Repair for Introductory Programming Courses. https://doi.org/10.48550/arXiv.2107.06550 arXiv:2107.06550 [cs].
[31] Siqi Ma, Ferdian Thung, David Lo, Cong Sun, and Robert H. Deng. 2017. VuRLE: Automatic Vulnerability Detection and Repair by Learning from Examples. In *Computer Security – ESORICS 2017 (Lecture Notes in Computer Science)*, Simon N. Foley, Dieter Gollmann, and Einar Snekkenes (Eds.). Springer International Publishing, Cham, 229–246. https://doi.org/10.1007/978-3-319-66399-9_13
[32] Martin Monperrus. 2018. *The Living Review on Automated Program Repair.* Technical Report hal-01956501. HAL Archives Ouvertes.
[33] Adib Nahiyan, Jungmin Park, Miao He, Yousef Iskander, Farimah Farahmandi, Domenic Forte, and Mark Tehranipoor. 2020. SCRIPT: A CAD Framework for Power Side-channel Vulnerability Assessment Using Information Flow Tracking and Pattern Generation. *ACM Transactions on Design Automation of Electronic Systems* 25, 3 (May 2020), 26:1–26:27. https://doi.org/10.1145/3383445
[34] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. https://doi.org/10.48550/arXiv.2203.13474 arXiv:2203.13474 [cs].
[35] OpenAI. 2021. OpenAI Codex. https://openai.com/blog/openai-codex/
[36] Jonas Oppenlaender. 2022. A Taxonomy of Prompt Modifiers for Text-To-Image Generation. https://doi.org/10.48550/arXiv.2204.13988 arXiv:2204.13988 [cs].
[37] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2022. Examining Zero-Shot Vulnerability Repair with Large Language Models. IEEE Computer Society, 1–18. https://doi.org/10.1109/SP46215.2023.00001
[38] Hammond Pearce, Benjamin Tan, and Ramesh Karri. 2020. DAVE: Deriving Automatically Verilog from English. In *2020 ACM/IEEE 2nd Workshop on Machine Learning for CAD (MLCAD)*. 27–32. https://doi.org/10.1145/3380446.3430634
[39] Nachiketh Potlapally. 2011. Hardware security in practice: Challenges and opportunities. In *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*. 93–98. https://doi.org/10.1109/HST.2011.5955003
[40] Pemma Reiter, Hui Jun Tay, Westley Weimer, Adam Doupé, Ruoyu Wang, and Stephanie Forrest. 2022. Automatically Mitigating Vulnerabilities in x86 Binary Programs via Partially Recompilable Decompilation. http://arxiv.org/abs/2202.12336 arXiv:2202.12336 [cs].
[41] Anonymized for review. 2023. Artifacts for "Large Language Models Can Fix Hardware Security Bugs". https://doi.org/10.5281/zenodo.7540216 Type: dataset.
[42] Masoud Rostami, Farinaz Koushanfar, and Ramesh Karri. 2014. A Primer on Hardware Security: Models, Methods, and Metrics. *Proc. IEEE* 102, 8 (Aug. 2014), 1283–1295. https://doi.org/10.1109/JPROC.2014.2335155
[43] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/2491956.2462195
[44] Hendrik Strobelt, Albert Webson, Victor Sanh, Benjamin Hoover, Johanna Beyer, Hanspeter Pfister, and Alexander M. Rush. 2023. Interactive and Visual Prompt Engineering for Ad-hoc Task Adaptation with Large Language Models. *IEEE Transactions on Visualization and Computer Graphics* 29, 1 (Jan. 2023), 1146–1156. https://doi.org/10.1109/TVCG.2022.3209479 Conference Name: IEEE Transactions on Visualization and Computer Graphics.
[45] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2022. Benchmarking Large Language Models for Automated Verilog RTL Code Generation. https://doi.org/10.48550/arXiv.2212.11140 arXiv:2212.11140 [cs].
[46] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. 2021. Fuzzing Hardware Like Software. https://doi.org/10.48550/arXiv.2102.02308 arXiv:2102.02308 [cs].
[47] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Transactions on Software Engineering and Methodology* 28, 4 (Sept. 2019), 19:1–19:29. https://doi.org/10.1145/3340544
[48] Aakash Tyagi, Addison Crump, Ahmad-Reza Sadeghi, Garrett Persyn, Jeyavijayan Rajendran, Patrick Jauernig, and Rahul Kande. 2022. TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities. *arXiv:2201.09941 [cs]* (Jan. 2022). http://arxiv.org/abs/2201.09941 arXiv: 2201.09941.
[49] vclint. 2022. Synopsys VC SpyGlass Lint. https://www.synopsys.com/verification/static-and-formal-verification/vc-spyglass/vc-spyglass-lint.html
[50] Weichao Wang, Zhaopeng Meng, Zan Wang, Shuang Liu, and Jianye Hao. 2019. LoopFix: an approach to automatic repair of buggy loops. *Journal of Systems and Software* 156, C (Oct. 2019), 100–112. https://doi.org/10.1016/j.jss.2019.06.076
[51] Chu-Pan Wong, Priscila Santiesteban, Christian Kästner, and Claire Le Goues. 2021. VarFix: balancing edit expressiveness and search effectiveness in automated program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 354–366. https://doi.org/10.1145/3468264.3468600
[52] Jiang Wu, Zhuo Zhang, Deheng Yang, Xiankai Meng, Jiayu He, Xiaoguang Mao, and Yan Lei. 2022. Fault Localization for Hardware Design Code with Time-Aware Program Spectrum. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*. 537–544. https://doi.org/10.1109/ICCD56317.2022.00085 ISSN: 2576-6996.
[53] Ying Zhang, Ya Xiao, Md Mahir Asef Kabir, Danfeng, Yao, and Na Meng. 2022. Example-Based Vulnerability Detection and Repair in Java Code. https://doi.org/10.48550/arXiv.2203.09009 arXiv:2203.09009 [cs].
[54] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large Language Models Are Human-Level Prompt Engineers. http://arxiv.org/abs/2211.01910 arXiv:2211.01910 [cs].

APPENDIX

Compute environment

All experiments were conducted on a Intel Core i5-10400T CPU @2GHzx12 processor with 16 GB RAM. Operating system Ubuntu 20.04.5 LTS was used.

Open source details

There are a few parts of our experimental framework where we could not provide fully open-source access:

- Verific: We used Verific libraries provided by Verific under an academic license. Please contact Verific to get access to their products.
- CWEAT: We requested CWEAT code from the authors of the paper "Don't CWEAT It: Toward CWE Analysis Techniques in Early Stages of Hardware Design" [5]. The paper is available at https://dl.acm.org/doi/abs/10.1145/3508352.3549369. Please contact the authors for use/help with their codebase.
- CirFix: We used the CirFix benchmarks and results provided in the open-source github repository provided by the authors of the paper "CirFix: automatically repairing defects in hardware design code." [6] https://github.com/hammad-a/verilog_repair. Please contact the authors about use of their tools. Their paper is available at https://dl.acm.org/doi/10.1145/3503222.3507763.
- Hack@DAC SoC: We use the SoC used during the 2021 competition. Please contact them at info@hackatevent.org for more information/access.

https://t.me/learningnets