On Hardware Security Bug Code Fixes By Prompting Large Language Models

Baleegh Ahmad, *Graduate Student Member, IEEE*, Shailja Thakur, *Member, IEEE*, Benjamin Tan, *Member, IEEE*, Ramesh Karri, *Fellow, IEEE*, and Hammond Pearce, *Member, IEEE*

***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 may be leveraged to automatically repair identified security-relevant bugs present in hardware designs by generating replacement code. We focus on bug repair in code written in Verilog. For this study, we curate 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 fifteen of our benchmarks. This ensemble outperforms a state-of-the-art automated hardware bug repair tool on its own suite of bugs. These results show that LLMs have the ability to repair hardware security bugs and the framework is an important step towards the ultimate goal of an automated end-to-end bug repair tool.**

***Index Terms*— Hardware security, large language models, bug repair.**

I. 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 [1]. While considerable effort has explored software repair, for Hardware Description Languages (HDLs), the state of the art is less mature.

In this study, we focus on repairing security-relevant hardware bugs after a bug has been identified by some means. While there are several approaches and tools for *detecting* potential design bugs [2], [3], [4], [5], [6], [7], few techniques address the automated repair of hardware bugs. The recently proposed CirFix [8] 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. 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.

Large Language Models (LLMs) are neural networks trained over millions of lines of text and code [9]. LLMs that are fine-tuned over open-source code repositories can generate code, where a user "prompts" the LLM with some text (e.g., code and comments) to guide the code generation. In contrast to previous 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 that could draw from a designer's expertise. 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. One distinction is that LLMs do not need an oracle to *generate* a repair, although one could be useful for evaluating whether the generated repair was successful. LLMs may generate a few potential fixes, leaving the designers to choose which repair is optimal. In prior work [10], [11], LLMs have been used to generate functional Verilog code. Machine learning-based techniques such as Neural Machine Translation [12] and pre-trained transformers [13] are explored in the software domain for bug fixes. Xia et al. [14] use LLMs to successfully generate repairs for software bugs in 3 different languages. Pearce et al. [15] 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's GPT (generative pre-trained transformer), Codex and CodeGen LLMs on instances of hardware security bugs. We offer insights into how best to use LLMs for successful repairs. A Register Transfer Level (RTL) designer can spot a security weakness and the LLM can help to find a fix. Our contributions are as follows:

- An automated framework for using LLMs to generate repairs and evaluate them. We make the framework and artifacts produced in this study available as open-source [16] (further details also included in the Appendix).

Manuscript received 3 July 2023; revised 21 December 2023 and 5 February 2024; accepted 6 February 2024. Date of publication 7 March 2024; date of current version 2 May 2024. This work was supported in part by Intel Corporation and in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) under Grant RGPIN-2022-03027. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Stjepan Picek. *(Corresponding author: Baleegh Ahmad.)*

Baleegh Ahmad, Shailja Thakur, and Ramesh Karri are with the Department of Electrical and Computer Engineering, New York University Tandon School of Engineering, Brooklyn, NY 11201 USA (e-mail: ba1283@nyu.edu).

Benjamin Tan is with the Department of Electrical and Software Engineering, University of Calgary, Calgary, AB T2N 1N4, Canada.

Hammond Pearce is with the Department of Electrical and Computer Engineering, University of New South Wales, Sydney, NSW 2052, Australia.

Digital Object Identifier 10.1109/TIFS.2024.3374558

- An exploration of bugs and LLM parameters (model, temperature, prompt) to see how to use LLMs in repair. They are posed as research questions answered in Section V.
- A demonstration of how the repair mechanism could be coupled with bug detectors to form an end-to-end solution to detect and repair bugs. This is presented in Section VII.

II. Background and Related Work

To provide context for our work, we discuss some overarching concepts in Section II-A. We present some differences between our work and other efforts in Section II-B.

A. Background

The code repair problem is well-explored in the software domain. Software code repair techniques continue to evolve (interested readers can see Monperrus' living review [17]). Generally, techniques try to fix errors by using program mutations and repair templates paired with tests to validate any changes [18], [19]. Feedback loops are constructed with a reference implementation to guide the repair process [20]. Other domain-specific tools may also be built to deal with areas like build scripts, web, and software models.

Security bugs are defects that can lead to vulnerable systems. While functional bugs can be detected using classical testing, security bugs are more difficult to detect, and proving their presence or absence is challenging. This has led to more "creative" kinds of bug repair, including AI-based machine-learning techniques such as neural transfer learning [21] and example-based approaches [22]. ML-based approaches, including neural networks, allow a greater ability to suggest repairs for "unseen" code. Example-based approaches start off with a dataset comprising 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 [23]. These approaches give credence to the ability of neural networks to learn from a larger set of correct code and inform repair on an instance of incorrect code.

We focus on hardware bugs originating at the Register-Transfer Level (RTL) stage. RTL designs, typically coded in 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) changes their outputs near-instantaneously according to the inputs. While software code describes programs that will be executed from beginning to end, RTL specified in HDL describes components that 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 [24]. 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 also a concern, and the synthesis and optimization of secure circuits starts with the description of designs with HDLs [25].

Linters [26], [27] and formal verification tools [3], [4] cover a large proportion of *functional* bugs. Although formal verification tools like Synopsys FSV can be used for security verification in the design process, they can sometimes have limited success [28]. With the ever-growing complexity of modern processors, software-exploitable hardware bugs are becoming pernicious [29], [30]. This has resulted in the exploration of many detection techniques such as fuzzing [5], information flow tracking [6], unique program execution checking [7] and static analysis [2].

Security-related issues that arise because of bugs in hardware are taxonomized in the form of Common Weakness Enumerations (CWEs). MITRE [31] is a not-for-profit that works with academia and industry to develop a list of 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. We select a subset of the hardware CWEs based on their clarity of explanation and relevance to RTL. A large subset of CWEs are not related to the RTL as they cover post-silicon issues or using outdated technologies or firmware. For some of the remaining CWEs, their descriptions can be vague and imprecise, making it difficult to reason about their presence with a great degree of confidence. In this work, we focus on the following CWEs:

1234: Hardware Internal or Debug Modes Allow Override of Locks. System configuration controls, e.g., memory protection are 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, it is useless.

1276: Hardware Child Block Incorrectly Connected to Parent System. If an input is incorrectly connected, it affects security attributes like resets while maintaining correct function and 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 functions in different states. When FSMs are used in modules that control the level of security a system is in, it is important that the FSM does not have any undefined states. These 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.

1298: Hardware Logic Contains Race Conditions. Race conditions may result in a timing error or glitch that causes the output to enter an unknown or unwanted state before settling to its desired value. If this happens in access control logic or sensitive security flows, an attacker may use it to bypass protections.

1231: Improper Prevention of Lock Bit Modification. A trusted lock bit may be used to restrict access to registers, address regions, device configuration controls and other resources after their values are assigned. If the lock bit can be modified after assignments to these resources, attackers may be able to access and modify the assets the lock bit is trying to protect.

1311: Improper Translation of Security Attributes by Fabric Bridge. A bridge allows IP blocks supporting different fabric protocols to be integrated into the system. If the translation of security attributes in this bridge is incorrect, the identity of an IP could be ascribed as trusted as opposed to trusted. This exposes the system to control bypass, privilege escalation or denial of service.

1254: Incorrect Comparison Logic Granularity. Comparison logic is used for purposes like password checks. If the comparison is done over a series of steps where the comparison failure at one of the steps breaks the operation, it may be vulnerable to a timing attack.

1224: Improper Restriction of Write-Once Bit Fields. Hardware design control registers have to initialized at reset to defined values. To prevent modification by software, they can be written into only once, after which they become read-only. Failure to implement write-once restrictions allows them to be modifiable.

B. Related Work

CirFix [8] attempts to localize bugs in RTL designs and then repair them. CirFix uses an iterative stochastic search with an instrumented testbench that captures the behavior of the circuit. This testbench is an oracle that requires the input parameters and expected outputs. Another related work is VeriSketch [32]. Here, synthesis, information flow tracking, and code-repair techniques are coupled with function and security properties to produce secure RTL. While Verisketch does borrow repair-oriented idea of iteratively generating RTL code till it is secure, it is a secure-by-design approach.

The differences between our work with CirFix and VeriSketch are outlined in Table I. CirFix performs localization/identification of the bug and the repair. These two parts can be examined independently, e.g., Tarsel [6] uses hardware-specific timing information and the program spectrum and captures the changes of executed statements to locate faults effectively. VeriSketch in contrast aims at neither localization nor repair; it uses repair techniques to generate designs that are secure according to the properties provided by security architects/designers.

TABLE I
COMPARISON WITH CIRFIX AND VERISKETCH

	CirFix [8]	VeriSketch [32]	LLMs (this work)
Goal	Localization and repair	Secure by design	Repair only
Requires	Repair templates and operators	Security properties	Instructions
Oracle?	Yes	Yes	No
Iterative?	Yes	Yes	No

In our work, we focus on *repair*, using LLM-generated replacement code. We leverage some of the designer's expertise, and examine how much can be achieved with LLM-based repair, where generation does not need an "oracle". While CirFix instruments an oracle to use the correct outputs to guide repairs, LLMs rely on the many RTL code from training to produce a corrected version of the buggy code. VeriSketch requires functional properties to generate the design which functions as an oracle. We present an empirical comparison of LLM-based replacement code to CirFix in Section VI.

The first work that used LLMs to repair code (C,Python and Verilog) was done by Pearce et al. [15]. For Hardware, the authors covered 2 CWEs by looking at 2 bug instances in simple designs. We build on their basic idea by focusing specifically on hardware and covering more bugs and CWEs to gain more insights about using LLMs for repair of RTL. We cover 10 CWEs, 15 security related bugs and explore functional repair capabilities of LLMs as well by comparing performance of LLMs with CirFix. We also combine the repair ability of LLMs with the static hardware bug detector tool CWEAT [2], to show how a bug can be detected and repaired using our approach.

III. DESIGNS AND BUGS

To explore using LLMs to fix HW security bugs, we collate and prepare a set of 15 hardware security bug benchmark designs from three sources: CWE descriptions on the MITRE website [31], OpenTitan System-on-Chip (SoC) [33] and the Hack@DAC 2021 SoC [34]. The bugs from MITRE and the Hack@DAC 2021 SoC were available previously. We inserted bugs into the OpenTitan designs. Each bug is represented in a design, as described in Table II.

A. MITRE's CWE Examples

We use examples in MITRE's hardware design list to develop simple designs representing CWE(s). 4 of the 9 bugs and corresponding fixes for this source are shown in Figure 1 and detailed below. Bugs 5-9 are described in Table II but their coded examples are skipped for brevity. Their details may be found at [16].

TABLE II

BUGS OVERVIEW. WE ASSIGN A CWE TO EACH BUG AND GIVE A DESCRIPTION OF THE DESIGN

	Design	CWE	Source	Inserted by Authors?	Description
1	Locked Register	1234	MITRE	No	A lock mode blocks writes after lock is set to 1. However, the lock protection is overridden when debug unlocked mode is active.
2	Lock on Reset	1271	MITRE	No	A locked register does not have a value assigned on reset. When the circuit is brought out of reset, the state is unknown.
3	Grant Access	1280	MITRE	No	This module modifies register contents 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	No	A peripheral is instantiated within a SoC using a signal to indicate trusted and untrusted entities. This signal depicting the security level is incorrectly grounded.
5	Mux	1298	MITRE	No	2x1 Multiplexor using logic-gates. There is a glitch in the output signal when value of control signal changes with input signals.
6	OCP2AHB Bridge	1311	MITRE	No	The bridge interfaces between OCP and AHB end points. The OCP protection signal incorrectly translates untrusted IDs as trusted to the AHB protection signal.
7	Reglk Wrapper	1231	MITRE	No	In the reset controller module, a peripheral reset signal with user privilege can reset the register locks.
8	Password Check	1254	MITRE	No	This module checks a user-provided password in a byte-by-byte manner, terminating when any byte fails. This comparison is vulnerable to a timing attack.
9	Write Once Register	1224	MITRE	No	A register has a write-once field defined by a write once status bit. This bit is dependent on input data, making the write once attribute insecure.
10	ROM Control	1245	OpenTitan	Yes	An alert should be triggered in the FSM if start signal is high in any state other than Waiting. The state is incorrectly compared to the Done instead of Waiting.
11	OTP Control	1245	OpenTitan	Yes	The life cycle interface FSM should move into an invalid state upon global escalation.The corresponding error signal for this scenario is not asserted.
12	Keymanager KMAC	1245	OpenTitan	Yes	The FSM has a done signal which should only be asserted at completion. This signal is asserted outside of expected window, i.e., during a transmission state.
13	CSR RegFile	1234	H@DAC-21	No	The core should be unstalled on a pending or incoming interrupt. The core is also incorrectly unstalled if there is a request to enter debug mode.
14	DMA	1271	H@DAC-21	No	A register 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.
15	AES-2 interface	1245	H@DAC-21	No	The FSM has a total of 15 states and does not include a default statement for its 4 bitstate variable. This represents an incomplete case statement of an FSM.

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 1(a), a `debug_unlocked` signal overrides the `lock_status` signal allowing the locked register to be written into even if `lock_status` is asserted.

2) Lock on Reset: design has a register that holds sensitive information. This register should be assigned a known value on reset. In Figure 1(b), ttlocked register should have a value on reset, but in this case, there is no reset.

3) Grant Access: This design has a register that should only be modifiable if the `usr_id` input is correct. In Figure 1(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.

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 1(d), the security level of the instantiated peripheral is grounded to zero, which could lead to incorrect privilege escalation of all input data.

B. Google's OpenTitan

OpenTitan is an open-source project to develop a silicon root of trust with implementations of SoC security measures. Since OpenTitan does not have declared bugs, we inject bugs by tweaking the RTL of these security measures in different modules. These are measures implemented in the HDL code that mitigate attacks on assets in security-critical Intellectual Properties (IPs). The OpenTitan taxonomy presents a countermeasure in the following form: `[UNIQUIFIER.]ASSET.CM_TYPE`. Here, `ASSET` is the element that is being protected e.g., key or internal states of a processor's control flow. Each protection mechanism is named with `CM_TYPE` e.g., multi-bit encoded signal, scrambled asset, or access to asset limited according to life-cycle state. The `UNIQUIFIER` is a custom prefix label to make the identifier unique after identifying the IP. The bugs we produced using these countermeasures and their corresponding fixes are shown in Figure 2.

1) ROM Control: This design contains a module that acts as an interface between the Read Only Memory (ROM) and the system bus. The ROM has scrambled contents, and the controller descrambles the content while serving memory requests. We target the `COMPARE.CTRL_FLOW. CONSISTENCY` security measure in the `rom_ctrl_compare` module. Here, the asset is `CTRL_FLOW` referring to the control flow of the ROM Control module. The countermeasure is `CONSISTENCY`, checking consistency of the control flow other than by associating integrity bits. 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 2(a), because of our induced bug, the alert signal is incorrectly asserted when `start_i` is high in any state other than `Waiting`.

```
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) Data_out <= 16'h0000;
else if(write&(~lock_status|debug_unlocked))begin
else if (write&~lock_status) begin
    Data_out <= Data_in;
end
else if (~write) Data_out <= Data_out;
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
     ( 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.

Fig. 1. MITRE CWE bugs and repairs. A repair (green) replaces the bug (red).

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 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. Here, the asset is `FSM` referring to the Finite State Machine of the OTP Control module. The countermeasure is `LOCAL_ESC`, ensuring a trigger when an attack is detected. A part of this measure is that the FSM jumps to an error state if the escalation signal is asserted. In Figure 2(b), no error is raised in such a case because of our induced bug.

```
logic start_alert;
assign start_alert=start_i && (state_q != Done);
assign start_alert=start_i && (state_q != Waiting);
```

(a) 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)) state_d = StTxLast;
end
```

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

Fig. 2. OpenTitan bugs: The repair (green) replaces the bug (red) for a successful fix.

3) Keymanager KMAC: This design performs 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. Here, the asset is `CTRL`, referring to the logic used to steer the hardware behavior of the KMAC module. The countermeasure is `CONSISTENCY`, checking the consistency of the control hardware other than by associating integrity bits. 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 2(c), because of our induced bug, the kmac done signal is incorrectly asserted in the transmission state `StTx`.

C. Hack@DAC-21

This is a hackathon for finding vulnerabilities at the RTL level for an System-on-Chip (SoC). The bugs and fixes for this source are shown in Figure 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 function pertaining to the stalling of the core on receiving an interrupt and/or debug request. In Figure 3(a), the debug signal overrides interrupt signals.

2) DMA: design has Direct Memory Access module common to all blocks. It uses the memory address as input and performs read/write according to the Physical Memory Protection (PMP) configuration. We consider PMP access mechanism as the relevant security implementation.

```
// 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)
 wfi_d=1'b1;
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.

Fig. 3. Hack@DAC bugs: Repair code (green) replaces buggy code (red).

In Figure 3(b), the PMP register is not assigned any value on reset.

3) AES 2 Interface: This design incorporates the AES module to produce cipher text while using an FSM for managing interactions. However, the case statement in Figure 3(c) lacks enough cases and does not include a default statement.

IV. Experimental Method

To test the capability for LLMs to generate successful repairs, we perform experiments using the designs and bugs detailed in Section III. In this section, we present our framework that automates the execution of our experiments, starting from the location of bugs to the evaluation of the repairs.

A. LLM-Based Repair Evaluation Framework

Figure 4 provides an overview of our experiments, with three main components, the **Sources**, **Repair Generator**, and **Evaluator**. The Sources were discussed in Section III.

1) Repair Generator: This block takes the **buggy file**, **location** and **CWE** of the bug as the input from the source. We assume that the location of the bugs, i.e., starting and ending line numbers and the filepath of the buggy file, is known. The location is only used to construct the content window for the prompt and is not a feature learned by the LLM. For each bug, we develop *instructions to assist the repair*: these are in the form of comments, which will be inserted before and after the buggy code to 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 for bug in Figure 1(c) is shown in Figure 5 (a)-(b). 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 IV-B.1. The **LLM** takes the **Prompt** as input and outputs the **Repairs**. The repairs produced may be correct or incorrect. Some of the repairs generated using the prompt Figure 5 (b) are shown in Figure 5 (c)-(e).

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 [35] as a part of Xilinx Vivado 2022.2 to simulate the designs. **Functional Evaluation** uses Verilog testbenches we made for each design comprising of tests 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 another step of forming the Device Under Test (DUT) before simulation. We prepare for the simulator a list of files instantiated by the buggy file and the files that need to be analyzed prior to the buggy file.

Security Evaluation involves a combination of testbenches (for MITRE and OpenTitan) and a static analysis tool (for Hack@DAC) discussed in Section VII. For MITRE designs, we design tests based on weaknesses mentioned on the MITRE website for each bug. For OpenTitan, we use the security countermeasures defined in relevant ".hjson" files for the peripherals. It is difficult to verify the security countermeasure completely because that requires simulating the SoC through the software for Design Verification by OpenTitan. This method is a work in progress for the OpenTitan team. Moreover, the countermeasures that can currently be verified completely require a lot of simulation time. Hence, we develop custom testbenches that verify specific functionality for the bugs we introduce in the OpenTitan.

B. Experimental Parameters

LLMs have several controllable parameters which effect output generation. We change the prompt (as discussed in Section IV-A.1) according to the bug and **Instructions**. We also vary the **Temperature** and **Models** while keeping the **top_p**, **number_of_completions (n)** and **max_tokens** constant at 1, 20 and 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 [9].

1) Instruction Variation: We test five instruction variants to guide the repair of bugs. They are described in Table III. Each

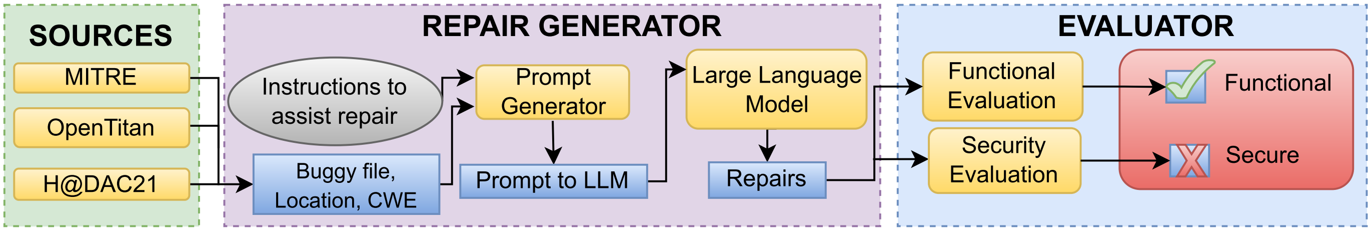


Fig. 4. Our experimental framework has 3 components: Sources with bugs, LLM-based Repair Generator to create fixes, and Evaluator to verify repairs.

```
// BUG: Access Control Check Implemented
// after asset is Accessed.
// Ensure that access is granted before data
// is accessed.
// FIX:
```

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

```
module user_grant_access
     ( 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:
```

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

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

(c) Correct repair produced by code-davinci-001 at temp=0.7. Signal grant_access is assigned a value based on user id before it is checked.

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

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

```
if (grant_access) begin
    data_out = data_in;
end
```

(e) Incorrect repair produced by code-davinci-001 at temp=0.7. This response failed the functional and security evaluations. Signal grant_access is not assigned any value, so it remains in an unknown state.

```
module user_grant_access
     ( 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) begin
        data_out = data_in;
    end
end
end
endmodule
```

(f) Generated repair file using Figure 5(c). This is sent for evaluation.

Fig. 5. Prompt to LLM and sample repairs produced for Bug 3 - Grant Access. Sub-figures (a)-(b) show how the bug is combined with instructions to generate the prompt that the LLM gets as one of its inputs. Sub-figures (c)-(e) show some actual repairs generated by an LLM.

TABLE III

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 and prescriptive instruction on how to fix
d	Natural language description of bug and descriptive instruction on how to fix
e	Code examples of bug and fix

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. Here, 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. This is an attempt to increase generalizability of some variations of prompts allowing the repair of different instances of the same 'kind/CWE' of bug. For 9 of the 10 CWEs, we use the description of the CWE as the Bug Instruction for variations *b, c, d and e* . We make an exception for CWE 1245 because it covers a vast possibility of issues. CWE 1245 is "Improper FSMs in Hardware Logic" which may include incomplete case statements, vulnerable transitions, missing transitions, FSM deadlocks, incorrect alert mechanisms etc. 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 IV.

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

3) Models: We use seven LLMs, five of which are made available by OpenAI [36] and two are open-source models available through [10] and [37]. The OpenAI Codex models are derived from GPT-3 and were trained

TABLE IV

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: Hardware Logic Contains Race Conditions	// assign y = a|b;
		// FIX:
		assign y = a | b | (in1 & in2);
			/// The value assigned to signal should remain in a continuous known state without any glitch.	// include input signals in assignment to avoid glitch	‘\n’

6	// BUG: Improper Translation of Security Attributes by Fabric Bridge	// case (temp[3:1]); 3'b000: a = 2'b11;
		3'b001: a = 2'b00; 3'b010: a = 2'b00;
		3'b011: a = 2'b01; 3'b100: a = 2'b00;
		...
		// FIX:
		case (temp[4:0]); 5'h11: a = 2'b10;
		5'h10: a = 2'b00; 5'h0F: a = 2'b00;
		5'h0D: a = 2'b10; 5'h0C: a = 2'b00;
		5'h0B: a = 2'b00; 5'h09: a = 2'b10;
		...
			// Translate the appropriate security bits from one protocol to the other.	// Select the appropriate bits from intermediate signal p0_mreqinfo_o_temp to translate its security attributes to output signal ahb_prot	‘endcase’

7	// BUG: Improper Prevention of Lock Bit Modification	// if(~(rst_n && ~unlock && ~rst_local))
		// FIX:
		if(~(rst_n && ~unlock))
			// Use a condition that prevents lock bits from being modified by external peripherals.	// Do not include local reset signal in reset condition	‘\n’

8	// BUG: Incorrect Comparison Logic Granularity	Skipped because its too long.	// Implement comparison logic that takes equal number of operations regardless of the values being compared.	// Compare the values such that all bytes of the password are checked before exiting the checks. Each byte of password corresponds to a bit of check_pass.	-

9	// BUG: Improper Restriction of Write-Once Bit Fields	// write_once_status <= d_in[0];
		// FIX:
		write_once_status <= 1;
			// register containing status of write once bit field should be set high when data has been written.	// set write once status register high	‘\n’

10	// 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’

11	// 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

12	// 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

13	// BUG: Hardware Internal or Debug Modes Allow Override of Locks.	// if (debug || interrupt)
		// FIX:
		if (interrupt)
			// Debug or scan signals should not be included in if condition	// unstall core when interrupt is high	‘\n’

14	// 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	-

15	// 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

on millions of public GitHub repositories. They can ingest and generate code, and also translate natural language to code. We use/evaluate `gpt-3.5-turbo`, `gpt-4`, `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`. It is an autoregressive LLM for program synthesis trained sequentially on The Pile and BigQuery. We also evaluate the fine-tuned version of CodeGen, trained over a Verilog corpus comprising of open-source Verilog code in GitHub repositories [10], referred to as VGen.

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 5 (b), this would be lines 1–5 (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. If the bug is too large, however, the lines before the bug and the bug may exceed the token limit of the LLM. Then the proposed scheme will not be able to repair it. In our work though, we did not run into this problem.

5) Stop Keywords: They are not included in the response. We developed a strategy that works well with the set of bugs. The default stop keyword is `endmodule`. Keywords used are in the column Stop keywords in Table IV.

V. 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 heatmaps in Figure 6. The maximum value for each element is 20, i.e., when all responses were successful repairs.

A. RQ1: Can Out-of-the-Box LLMs Fix Hardware Security Bugs?

Results show that LLMs can repair simple security bugs. `gpt-4`, `code-davinci-002`, and `code-cushman-001` yielded at least one successful repair for every bug in our dataset. `code-davinci-001`, `gpt-3.5-turbo`, `CodeGen` and `VGen` were successful for 14, 13, 11 and 10 out of 15 bugs. In total, we requested 52,500 repairs out of which 15,063 were correct, a success rate of 28.7%. The key here lies in selecting the best-observed parameters for each LLM. `code-davinci-002` performs best at variation *e*, *temp* 0.1 producing 69% correct repairs. `gpt-4`, `gpt-3.5-turbo`, `code-davinci-001`, `code-cushman-001`, `CodeGen` and `VGen` perform best at $(e, 0.5)$, $(d, 0.1)$, $(d, 0.1)$, $(d, 0.1)$, $(e, 0.3)$ and $(c, 0.3)$ with success rates of 67%, 44%, 53%, 51%, 17% and 8.3% respectively. Performance of these LLMs across bugs is shown in Figure 7.

B. RQ2: How Important Are Prompt Details?

The 5 instruction variations from *a* to *e* increase in the level of detail. Apart from CodeGen and VGen, the LLMs do better with more detail when generating a repair, as shown in Figure 8. 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" (designer intent using mostly natural language) leads to the best results. There is variation within LLMs for the best-observed instruction variation, e.g. `gpt-4`, `code-davinci-002` and `CodeGen` perform best at *e*. Excluding the results of `CodeGen` and `VGen`, because they perform very poorly, the success rates for variations *a-e* across OpenAI models increase by 20, 41, 11 and -14 % respectively for each successive variation. As an example, going from variation *a* to *b* yields 20% more successful repairs and going from *b* to *c* yields 41% more successful repairs. From these numbers, the most significant jump is going from *b* to *c*, showing the importance of including a **Fix Instruction** in the prompt. We also observe that a coded example of a repair in the form of variation *e* decreases the success rates of OpenAI LLMs. Instructions with natural language guidance do better than coded examples.

C. RQ3: What Bugs Appear Amenable to Repair?

The cumulative number of correct repairs for each bug for OpenAI LLMs is shown in Figure 9. Bugs 3 and 4 were the best candidates for repair with success rates of over 75%. These are examples from MITRE where the signal names indicate their intended purposes. For the **Grant Access** module, the signals of concern are `grant_access` and `usr_id` used in successive lines. LLMs preserved the 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 functional.

Bugs 5, 6 and 12 were the hardest to repair with success rates $< 10\%$. Bug 6 was the toughest to repair because of the specificity required for the correct case statement. A correct repair would require all 32 possibilities of the security signal to be correctly translated to the 4 possible values of the output security signal. Bug 5 was difficult to repair because the models refused to acknowledge that a glitch existed and kept generated the same code as the bug. On many occasions `gpt-4` produced the variations of the following comment accompanying the code it generated: "No bug here, already correct." Bug 12 was the only bug that required a line to be removed without replacement as a fix. Bugs 8 and 14 were moderately difficult to repair with success rates over 10 but less than 20%. Bug 8 proved difficult because of its complexity. The bug spanned 20 lines and a typical repair required 4 if statements. Bug 14 had the bug of a register

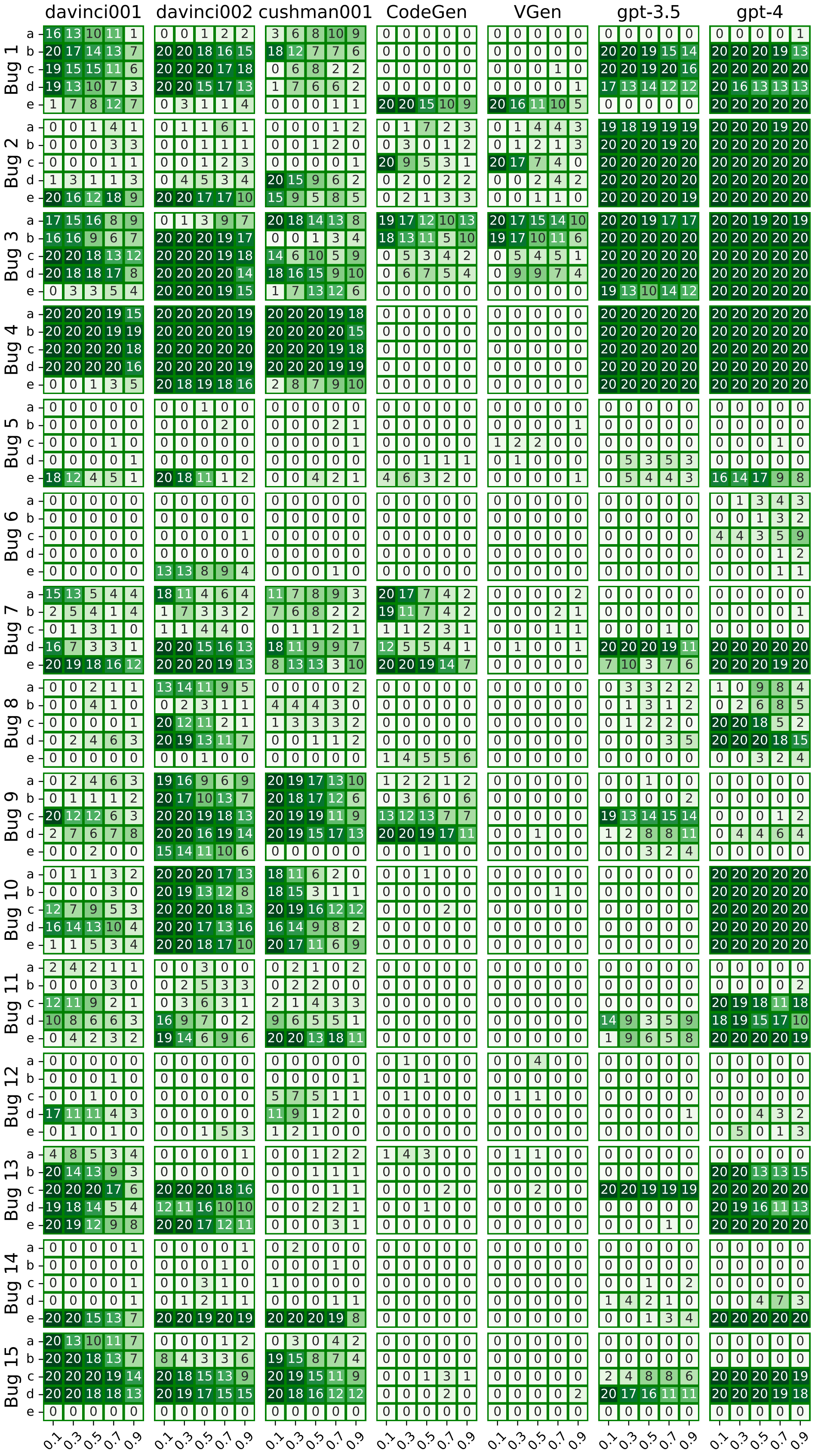


Fig. 6. Results for all LLMs, temperature and instruction variation configurations represented as heatmaps. The maximum value for each small box is 20. A higher value indicates more success by LLM in generating repair and is highlighted with a darker shade. All bugs were repaired at least once by at least one LLM.

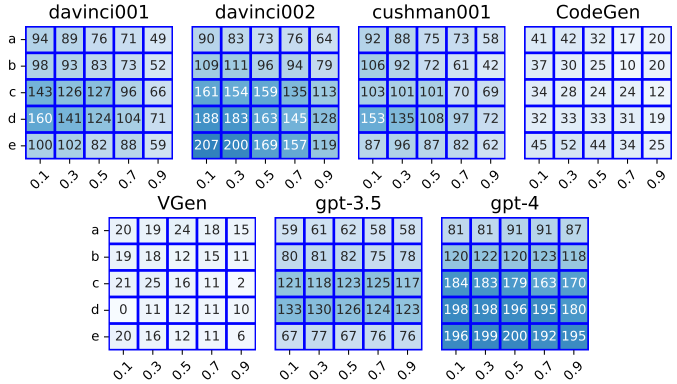


Fig. 7. Results showing the performance of each LLM across all bugs in the form of heatmaps. Each small square shows the number of correct repairs for the corresponding instruction variation and temperature of the LLM. The maximum possible value is 300. A higher value indicates more success in generating repairs and is shaded in a darker color.

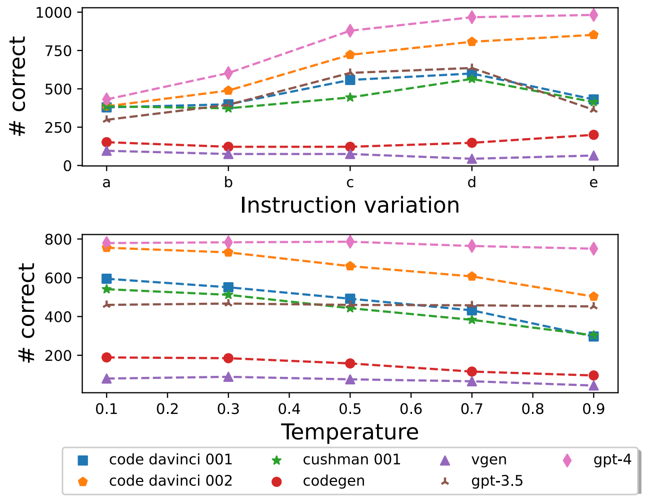


Fig. 8. Trends across models. The top graph shows the number of correct repairs for LLMs for specified instruction variations. The bottom graph shows the number of correct repairs for LLMs for specified temperature. The maximum value for each data point is 1500.

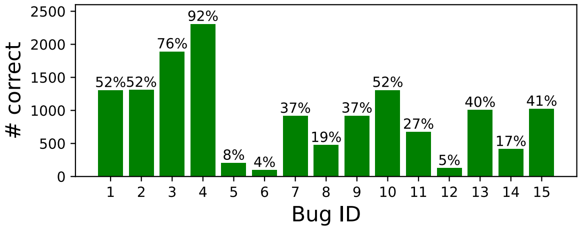


Fig. 9. Number of correct repairs per bug for OpenAI LLMs. The number above each bar shows the sum of successful repairs across all LLMs for the corresponding bug. The maximum possible value is 2500. A higher value indicates that the bug was repaired more times. Annotations represent the percentages of successful repairs.

holding security settings not initialized under reset. This was difficult to repair because a fix needs an added `always` block with an appropriate reset and re-creating the previous intended functionality.

D. RQ4: Does the LLM Temperature Matter?

A higher temperature allows the LLM to be more "creative." Overall, Figure 8 shows that the LLMs perform better at lower temperatures with a temperature of 0.1 yielding the highest number of successful repairs. The performance of most LLMs decreases with increasing temperature. For `gpt-4` and `gpt-3.5-turbo` however, temperature does not seem to have an impact with success rates being stable regardless of the temperature. `gpt-4` is the only LLM which performs best at a temperature of 0.5. `gpt-3.5` and `VGen` perform best at 0.3. The remaining perform best at 0.1. A lower temperature leads to less variation in responses, suggesting that the less creative responses are more likely to be correct repairs.

E. RQ5: Are Some LLMs Better Than Others?

The `gpt-4` LLM was the best performing, producing 3862 correct repairs out of 7500, giving it a success rate of 51.5%. `code-davinci-002`, `gpt-3.5-turbo`, `code-davinci-001`, `code-cushman-001`, `CodeGen` and `VGen` had success rates of 43.4%, 31.6%, 30.6%, 29.1%, 9.9% and 4.7% respectively. The difference between OpenAI LLMs and `CodeGen` + `VGen` is caused by `CodeGen` being a smaller LLM, with 16 billion parameters compared to the GPT-4's 1.7 trillion and GPT-3's ~175B parameters (the number of parameters for the OpenAI LLMs are not public). While the larger models tend to perform better, the relationship is not completely linear. For instance, while the code-davinci models and gpt-3.5-turbo are based on the same underlying GPT-3 model, the code-davinci-002 performs much better than gpt-3.5-turbo. This is because it is fine-tuned for use specifically for programming applications, whereas gpt-3.5-turbo has had additional fine-tuning to align it for instruction following. We believe that having a "large enough" model which has been fine-tuned appropriately would provide a suitable foundation for most applications. Counter-intuitively, the fine-tuned `VGen` performs worse than `CodeGen`. `code-cushman-001` is slightly inferior to `davinci` LLMs, possibly because it was designed to be quicker (smaller), i.e., it has fewer parameters, was trained over less data, or both.

VI. Comparison to Prior Work

To further validate the effectiveness of our approach, we performed a detailed comparison of our work with CirFix [8] (Table V). We use the best-performing LLM (`gpt-4`) at $t = 0.1$ and generate one repair each for instruction variations *a* and *b*.

Recall that a has no instruction, meaning the LLM is not guided at all. 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. Variation *a* produces 19 correct repairs as compared to CirFix's 16. To elicit the power of LLMs, we use variation *b* which includes a description of the type of bug. We use the brief descriptions of bugs provided in CirFix's GitHub repository. Variation *b* fixes 22 of the 32 benchmarks.

TABLE V

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 2 WAS REPAIRED

Project	**Defect Description**	**CirFix**	**LLM var** a	b
decoder (3 to 8)	Two numeric errors	y	y	y
	Incorrect assignment	-	-	-
first_counter overflow	Incorrect sensitivity list	y	-	-
	Incorrect increment of counter	y	y	y
	Incorrect reset	y	y	y
flip_flop	Incorrect conditional	y	-	-
	if-statement branches swapped	y	y	y
fsm_full	Incorrect case statement	-	-	-
	Assignment to next state and default in case statement omitted	y	y	y
	State omitted from senslist	-	-	y
	Incorrect blocking assignments	-	-	-
lshift_reg	Incorrect blocking assignments	y	-	y
	Incorrect conditional	y	y	y
	Incorrect sensitivity list	y	y	-
mux_4_1	Three numeric errors	-	y	y
	Hex instead of binary numbers	-	y	y
	1 bit instead of 4 bit output	-	y	y
i2c	Incorrect sensitivity list	y	y	y
	Incorrect address assignment	-	-	-
	No command acknowledgement	y	-	-
sha3	Off-by-one error in loop	y	-	y
	Incorrect assignment to wire	-	y	y
	Skipped buffer overflow check	y	-	-
	Incorrect bitwise negation	-	y	y
tate_pairing	Incorrect logic for bitshifting	-	y	y
	Incorrect instantiation of modules	-	y	y
	Incorrect operator for bitshifting	-	-	y
reed_solomon decoder	Insufficient register size for decimal values	-	y	y
	Incorrect sensitivity list for reset	y	y	y
sdram controller	Incorrect assignments to registers during synchronous reset	y	y	y
	Numeric error in definitions	-	y	y
	Incorrect case statement	-	-	-
		16	19	22

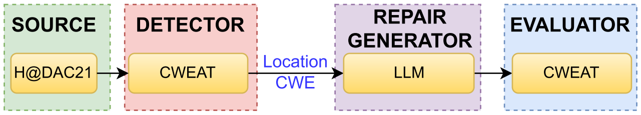


Fig. 10. Overview of the framework used in end-to-end solution with CWEAT. This was used for bugs 13, 14, and 15. The source is Hack@DAC 2021 SoC. CWEAT is used for both detection and evaluation.

VII. ON INCORPORATING A BUG DETECTOR

The work so far has discussed bug repair assuming that the location of the bug is already known. A detector could also be used after repair for security re-evaluation. To explore this possibility, we present an end-to-end framework for some CWEs in Verilog using CWEAT [2], which can detect a bug, generate a repair using our methodology, and then evaluate its correctness. This pipeline is shown in Figure 10.

CWEAT [2] is a static analysis tool that can detect some security weaknesses in RTL. We use the methods described in [2] to traverse the Abstract Syntax Trees (ASTs) generated by the Verific parser. 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. These are bugs 13, 14, and 15 in Table II. 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 (i.e. the bug was removed). **We produced results in Figure 6 for bugs 13, 14, and 15 using this end-to-end solution.**

We envision using this (or similar) LLM-infused end-to-end solution by RTL designers as they write HDL code in the early stages of Design. CWEAT can highlight weakness to the designer, run it through LLM to produce repairs, choose the ones that are secure, and present suggestions to the designer. Detection and repair can be treated as separate tasks and implemented using separate tools. A range of tools may be used for detection e.g., commercial linting tools, hardware fuzzers, information flow tracking, and formal verification. The bugs found may be repaired by methods including LLMs and CirFix. This hybrid approach is likely to detect the most bugs and produce the most successful repairs.

VIII. DISCUSSION AND LIMITATIONS

This study shows that LLMs have the potential for bug repair. Presently, some assistance is required from the designer to identify a bug's location/nature. This can be overcome by using tools to localize bugs and better design practices such as comments explaining functionality. Currently, designers may need to pick from options produced by LLMs. Static analysis tools like CWEAT can help select or refine fixes. Even LLMs can be used for identifying vulnerabilities. LLM4SecHW [38] is a LLM-based hardware debugging framework aiming to identify bugs and provide debugging suggestions during the hardware design iteration process. This however, requires curation of relevant data and fine-tuning using version control histories of relevant repositories.

We believe LLMs have the potential for automating bug repair. selecting the right LLM is the first and easiest choice. GPT-4 works the best because it has the largest number of parameters. Knowing some information about the bug significantly improves the performance. LLMs do not work well for certain kinds of bugs but do very well on others. Our work shows that GPT-4 has a success rate of 40% or more for 9 of the 15 bugs and 70% or more for 6. We observe that as the nature of the security bug becomes more "elusive," e.g., race conditions and incorrect comparison level granularity, LLMs perform significantly worse than on simpler bugs. A vulnerability could be present as a result of multiple bugs collectively producing a security issue. We hypothesize that these would be difficult to identify as they may be present in multiple locations. Additionally, in a production environment, we could use the repair mechanism within a validation loop. This is the primary reason we have included the coupling of a static detector with our approach in Section VII. So if the LLM is suggesting an incorrect repair, the detector will identify the repair as incorrect. In this scenario, if an LLM is able to produce even one correct repair for a bug, it will be able to find a successful fix.

A limitation of our study is the informal instruction variations. Although Bug instructions are inspired by the descriptions in CWEs, our Fix instructions are devised according to the experience of the authors. Our work reveals the importance of these variations, as subtle changes can affect the LLM response quality. Devising 5 categories is an attempt to systematize this process, but future work can explore more varieties. 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.

Another limitation of our study is that the functional and security evaluations are not exhaustive. Security evaluation is dependent on design-specific security objectives and cannot be exhaustive. With this in mind, we limit the security evaluation to the bug that makes the design insecure. 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. Ideal functional testbenches should be exhaustive for other examples too but are impractical. 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. It takes an hour to exhaustively simulate an IP on OpenTitan. Since we generate 3500 potential repairs for each bug, this would take 150 days for each bug. Therefore, we chose to build custom testbenches that test the code a repair could impact.

The choice of end-tokens 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. An LLM might not generate 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
```

if the keyword **end** is used as a stop token. Conversely, not limiting a response with an appropriate stop token may mean that the LLM produces the correct repair but then adds superfluous code. 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 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.

IX. Conclusion and Future Work

By selecting appropriate parameters and prompts, LLMs can effectively fix hardware bugs in our dataset. Each bug had at least one successful repair, and 13 of the 15 had perfect responses given the best parameter set. LLMs excel when signal names and comments indicate functionality but struggle with multi-line fixes or when a buggy line must be removed. Providing detailed, pseudo-code-like instructions improves repair success rates. Bigger LLMs and LLMs at lower temperatures perform better than those with fewer parameters and at higher temperatures. LLMs outperform CirFix in fixing function-related bugs in Verilog, even when detailed instructions are not provided. We suggest the following areas for future research:

- Employ a hybrid method for security bug detection with linters, formal verification, fuzzing, fault localization, and static analysis tools. For repair, use LLMs and oracle-guided modifying algorithms. Combining techniques is likely to yield better results than using just one.
- Fine-tune LLMs over HDLs and see if their performance improves. This improves quality of functional code [10].
- 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.

Appendix

Compute Environment

All experiments were conducted on an Intel Core i5-10400T CPU @2GHzx12 processor with 16 GB RAM, using Ubuntu 20.04.5 LTS.

Open Source Details

There are a few parts of our experimental framework where we could not provide fully open-source access. We used Verific libraries provided by Verific under an academic license. Please contact Verific for access.

We used CWEAT code from "Don't CWEAT It: Toward CWE Analysis Techniques in Early Stages of Hardware Design" [2]. 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.

We used the CirFix benchmarks and results in the GitHub repository provided by the authors of "CirFix: automatically repairing defects in hardware design code." [8] https://github.com/hammad-a/verilog_repair. Please contact the authors for their tools. Their paper is available at https://dl.acm.org/doi/10.1145/3503222.3507763.

We use the Hack@DAC SoC from the 2021 competition. Please contact them at info@hackatevent.org for more information/access.

Acknowledgment

The authors would like to thank Verific Design Automation for generously providing academic access to linkable libraries, examples, and documentation for their RTL parsers. This work does not in any way constitute an Intel endorsement of a product or supplier.

References

[1] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, pp. 56–65, Nov. 2019, doi: 10.1145/3318162.

[2] B. Ahmad et al., "Don't CWEAT It: Toward CWE analysis techniques in early stages of hardware design," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*. New York, NY, USA: Association for Computing Machinery, Oct. 2022, pp. 1–9.

[3] (2022). *VC Formal*. [Online]. Available: https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html

[4] Cadence. (2022). *Jasper RTL Apps Cadence*. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html

[5] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," 2021, *arXiv:2102.02308*.

[6] J. Wu et al., "Fault localization for hardware design code with time-aware program spectrum," in *Proc. IEEE 40th Int. Conf. Comput. Design (ICCD)*, Oct. 2022, pp. 537–544.

[7] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz, "Processor hardware security vulnerabilities and their detection by unique program execution checking," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 994–999.

[8] H. Ahmad, Y. Huang, and W. Weimer, "CirFix: Automatically repairing defects in hardware design code," in *Proc. 27th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.* New York, NY, USA: Association for Computing Machinery, Feb. 2022, pp. 990–1003, doi: 10.1145/3503222.3507763.

[9] M. Chen et al., "Evaluating large language models trained on code," 2021, *arXiv:2107.03374*.

[10] S. Thakur et al., "Benchmarking large language models for automated verilog RTL code generation," in *Proc. Design, Autom. Test Eur. Conf. Exhibition (DATE)*, Apr. 2023, pp. 1–6.

[11] H. Pearce, B. Tan, and R. Karri, "DAVE: Deriving automatically verilog from English," in *Proc. ACM/IEEE 2nd Workshop Mach. Learn. CAD (MLCAD)*, Nov. 2020, pp. 27–32.

[12] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 1–29, Sep. 2019, doi: 10.1145/3340544.

[13] D. Drain, C. Wu, A. Svyatkovskiy, and N. Sundaresan, "Generating bug-fixes using pretrained transformers," in *Proc. 5th ACM SIGPLAN Int. Symp. Mach. Program.*, Jun. 2021, pp. 1–8.

[14] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, May 2023, pp. 1482–1494.

[15] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2023, pp. 2339–2356.

[16] A. F. Rev. (2023). *Artifacts for 'On Hardware Security Bug Code Fixes By Querying Large Language Models'*. [Online]. Available: https://zenodo.org/records/10416865

[17] M. Monperrus, *The Living Review on Automated Program Repair*, document hal-01956501, HAL Arch. Ouvertes, 2018.

[18] W. Wang, Z. Meng, Z. Wang, S. Liu, and J. Hao, "LoopFix: An approach to automatic repair of buggy loops," *J. Syst. Softw.*, vol. 156, pp. 100–112, Oct. 2019, doi: 10.1016/j.jss.2019.06.076.

[19] X. D. Le and Q. L. Le, "ReFixar: Multi-version reasoning for automated repair of regression errors," in *Proc. IEEE 32nd Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2021, pp. 162–172.

[20] Y. Lu, N. Meng, and W. Li, "FAPR: Fast and accurate program repair for introductory programming courses," 2021, *arXiv:2107.06550*.

[21] Z. Chen, S. Kommrusch, and M. Monperrus, "Neural transfer learning for repairing security vulnerabilities in c code," *IEEE Trans. Softw. Eng.*, vol. 49, no. 1, pp. 147–165, Jan. 2023.

[22] S. Ma, F. Thung, D. Lo, C. Sun, and R. H. Deng, "VuRLE: Automatic vulnerability detection and repair by learning from examples," in *Computer Security—ESORICS*. Cham: Springer, 2017, pp. 229–246.

[23] P. Reiter, H. J. Tay, W. Weimer, A. Doupé, R. Wang, and S. Forrest, "Automatically mitigating vulnerabilities in binary programs via partially recompilable decompilation," 2022, *arXiv:2202.12336*.

[24] N. Potlapally, "Hardware security in practice: Challenges and opportunities," in *Proc. IEEE Int. Symp. Hardware-Oriented Secur. Trust*, Jun. 2011, pp. 93–98.

[25] D. Demmler, G. Dessouky, F. Koushanfar, A.-R. Sadeghi, T. Schneider, and S. Zeitouni, "Automated synthesis of optimized circuits for secure computation," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA: Association for Computing Machinery, Oct. 2015, pp. 1504–1517, doi: 10.1145/2810103.2813678.

[26] vclint. (2022). *Synopsys VC SpyGlass Lint*. [Online]. Available: https://www.synopsys.com/verification/static-and-formal-verification/vc-spyglass/vc-spyglass-lint.html

[27] Jasperlint. (2022). *Jasper Superlint App*. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/jaspergold-superlint-app.html

[28] G. Dessouky et al., "HardFails: Insights into software-exploitable hardware bugs," in *Proc. USENIX Secur. Symp.*, 2019, pp. 213–230.

[29] M. Lipp et al. (2018). *Meltdown: Reading Kernel Memory From User Space*. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[30] P. Kocher et al., "Spectre attacks: Exploiting speculative execution," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 1–19.

[31] T. M. Corp. (1194). *CWE—CWE-1194: Hardware Design (4.1)*. [Online]. Available: https://cwe.mitre.org/data/definitions/1194.html

[32] A. Ardeshiricham, Y. Takashima, S. Gao, and R. Kastner, "VeriSketch: Synthesizing secure hardware designs with timing-sensitive information flow properties," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 1623–1638, doi: 10.1145/3319535.3354246.

[33] (2019). *Hardware OpenTitan Documentation*. [Online]. Available: https://docs.opentitan.org/hw/

[34] HACK@EVENT. (2022). *HACK@DAC21—HacK@EVENT*. [Online]. Available: https://hackatevent.org/hackdac21/

[35] (2020). *ModelSim Vivado Design Suite Reference Guide: Model-Based DSP Design Using System Generator (UG958). Reader. AMD Adaptive Computing Documentation Portal*. [Online]. Available: https://docs.xilinx.com/r/en-U.S./ug958-vivado-sysgen-ref/ModelSim

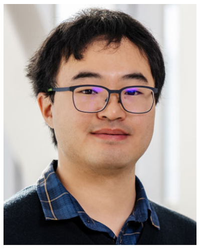
[36] OpenAI. (2021). *OpenAI Codex*. [Online]. Available: https://openai.com/blog/openai-codex/

[37] E. Nijkamp et al., "CodeGen: An open large language model for code with multi-turn program synthesis," 2022, *arXiv:2203.13474*.

[38] W. Fu, K. Yang, R. G. Dutta, X. Guo, and G. Qu, "LLM4SecHW: Leveraging domain-specific large language model for hardware debugging," in *Proc. Asian Hardw. Oriented Secur. Trust Symp. (AsianHOST)*, Dec. 2023, pp. 1–6.

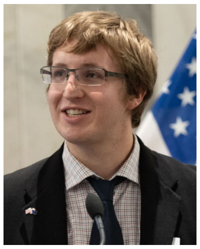
Baleegh Ahmad (Graduate Student Member, IEEE) received the B.Sc. degree in electrical engineering from New York University Abu Dhabi, Abu Dhabi, United Arab Emirates, in 2020. He is currently pursuing the Ph.D. degree in electrical and computer engineering with the New York University Tandon School of Engineering, Brooklyn, NY, USA. His research interests include hardware security and verification with a particular focus on bug detection and repair.

Shailja Thakur (Member, IEEE) received the Ph.D. degree from the University of Waterloo, Waterloo, ON, Canada, in 2022. She is currently a Post-Doctoral Research Associate with the Department of Electrical and Computer Engineering and the Center for Cybersecurity, New York University Tandon School of Engineering, Brooklyn, NY, USA. Her research interests include cyber-physical systems, electronic design automation, and large language models.



Benjamin Tan (Member, IEEE) received the B.E. degree (Hons.) in computer systems engineering and the Ph.D. degree from The University of Auckland, Auckland, New Zealand, in 2014 and 2019, respectively. In 2018, he was a Professional Teaching Fellow with the Department of Electrical and Computer Engineering, The University of Auckland. From 2019 to 2021, he was with New York University (NYU), Brooklyn, NY, USA, where he was a Post-Doctoral Associate and then a Research Assistant Professor, affiliated with the NYU Center for Cybersecurity. He is currently an Assistant Professor with the Department of Electrical and Software Engineering, University of Calgary, Calgary, AB, Canada. His research interests include computer engineering, hardware security, and electronic design automation.

Ramesh Karri (Fellow, IEEE) received the B.E. degree in electrical and computer engineering from Andhra University and the Ph.D. degree in computer science from the University of California San Diego. He is currently a Professor of electrical and computer engineering with New York University (NYU) Tandon School of Engineering. He co-directs the NYU Center for Cyber Security, co-founded the Trust Hub, and founded the Embedded Systems Challenge, the Annual Red Team Blue Team Event. He has published over 300 articles in leading journals and conference proceedings. His research and education in hardware cybersecurity include trustworthy ICs, processors, and cyber-physical systems; security-aware computer-aided design, test, verification, and nano meets security; hardware security competitions, benchmarks, and metrics; and additive manufacturing security.



Hammond Pearce (Member, IEEE) received the B.E. (Hons.) and Ph.D. degrees in computer systems engineering from The University of Auckland, Auckland, New Zealand. From 2020 to 2023, he was with New York University, Brooklyn, NY, USA, where he was a Post-Doctoral Research Associate and then a Research Assistant Professor with the Department of Electrical and Computer Engineering and the NYU Center for Cybersecurity. He is currently a Lecturer with the School of Computer Science and Engineering, University of New South Wales, Sydney, NSW, Australia. His research interests include cybersecurity of embedded and industrial systems, including additive manufacturing and industrial informatics.