Search+LLM-based Testing for ARM Simulators

1st Karine Even-Mendoza
Department of Informatics
King's College London
London, United Kingdom
karine.even_mendoza@kcl.ac.uk

2nd Héctor D. Menéndez
Department of Informatics
King's College London
London, United Kingdom
hector.menendez@kcl.ac.uk

3rd W.B Langdon
Department of Computer Science
University College London
London, United Kingdom
w.langdon@cs.ucl.ac.uk

4th Aidan Dakhama
Department of Informatics
King's College London
London, United Kingdom
aidan.dakhama@kcl.ac.uk

5th Justyna Petke
Department of Computer Science
University College London
London, United Kingdom
j.petke@ucl.ac.uk

6th Bobby R. Bruce
Department of Computer Science
University of California, Davis
United States of America
bbruce@ucdavis.edu

***Abstract*—In order to aid quality assurance of large complex hardware architectures, system simulators have been developed. However, such system simulators do not always accurately mirror what would happen on a real device. A significant challenge in testing these simulators arises from the complexity of having to model both the simulation and the infinite number of software that could be run on such a device.**

Our previous work introduced `SearchSYS`, a testing framework for software simulators. `SearchSYS` leverages a large language model for initial seed C code generation, which is then compiled, and the resultant binary is fed to a fuzzer. We then use differential testing by running the outputs of fuzzing on real hardware and a system simulator to identify mismatches.

We present and discuss our solution to the problem of testing software simulators, using `SearchSYS` to test the gem5 VLSI digital circuit simulator, employed by ARM to test their systems. In particular, we focus on the simulation of the ARM silicon chip Instruction Set Architecture (ISA).

`SearchSYS` can create test cases that activate bugs by combining LLMs, fuzzing, and differential testing. Using only LLM, `SearchSYS` identified 74 test cases that activated bugs. By incorporating fuzzing, this number increased by 93 additional bug-activating cases within 24 hours. Through differential testing, we identified 624 bugs with LLM-generated test cases and 126 with fuzzed test inputs. Out of the total number of bug-activating test cases, 4 unique bugs have been reported and acknowledged by developers. Additionally, we provided developers with a test suite and fuzzing statistics, and open-sourced `SearchSYS`[1].

***Index Terms*—Software System Simulation, `SearchSYS`, `gem5`, AFL++, Fuzzing, Fuzz Testing, Differential Testing, Search-Based Software Testing, LLM, Language Models, Ollama, CodeLlama, TinyLlama, Phi2, Llama2, Magicoder, CodeBooga, GPT-3.5-turbo, prompt engineering**

I. Introduction

Creating and developing new system architectures is a challenging task that requires a significant investment in both human and physical resources. Architectural simulators support this process by providing environments where developers can validate their architectural goals. A good example of these systems is `gem5` [1], [2], an open-source modular platform for computer system architecture research that includes system-level architecture and processor microarchitecture. The accuracy and reliability of software architecture simulators are paramount for optimizing development investment and ensuring a smooth lifecycle. However, software complexity of such simulators makes them difficult to test comprehensively for all potential architectural purposes. Identifying mismatches, internal simulation errors, and performance issues in simulators can lead to their improvement and make them more robust, thereby maximizing the return on investment.

Our previous work [3]–[6] addressed this problem by combining large language models (LLMs), fuzzing, and differential testing. By employing OpenAI's Large Language Model (LLM) `GPT-3.5-turbo` to generate a baseline test suite from seeded programs, and our modified version of AFL++, our system discovered various implementation bugs in `gem5` related to the Intel 64 bit X86 architecture. Whereas fuzzers traditionally rely on implicit test oracles, such as crashes, and timing out loops, differential testing strengthens the fuzzer by allowing it to automatically check for non-fatal errors, such as differences between running a simulation and running on actual hardware. We then extended this work to create `SearchSYS` [5], which fully automates our initial proof-of-concept and introduces novel mutation operators which are able to mutate not only the seed program itself, but also its command line parameters. Using `SearchSYS` we were able to further increase the bug finding ability of our approach. Again, we focused on the X64 architecture.

To showcase generalizability and widen impact and significance of our contribution, we now conduct an empirical study on the ARM family of CPUs. ARM is a $160 billion [7] company which designs chips and licenses them to others to manufacture and integrate them into devices. Although its CPU stretches the full gamut of today's computing, from supercomputing HPC, cloud computing, servers, desktops, and laptops, to Raspberry Pi, its processors are by far the most successful CPUs largely because almost all mobile telephones

[1] See https://zenodo.org/records/13450472 for initial seeds and `SearchSYS` code, and https://zenodo.org/records/13909721 for adjustment to ARM8v and fuzzed seeds.

and handheld smart devices are based on ARM CPUs. Altogether more than 200 billion ARM chips have been sold [8]. Naturally, such a company has considerable in-house expertise, nevertheless ARM makes use of the open source gem5 project keeping its own git clone but also paying close attention to fixes and developments to the public version [9], [10].

Despite being crucial to the quality of chip designs, at the cost of only a few days of computer time, by systematically using SearchSYS we were able to find and highlight 14 different types of issues with gem5's simulation of the ARM processor chips. Furthermore, SearchSYS has automatically created test cases for gem5's simulation of the ARM Instruction Set Architecture (ISA), which previously would have taken skilled engineers weeks to do by hand.

Finally, by simultaneously providing the gem5 development team with details of C++ source line coverage for each test, we open the way for future work whereby regression testing can be automatically targeted at immediate development changes. We expect that by executing tests that run the just modified code, there will be more chance of finding and addressing issues. Moreover, selecting which tests to run should speed up testing, possibly allowing real-time continuous integration testing [11]–[13]. Experience with Meta [14]–[16] highlighted the importance of giving individual developers immediate feedback. Hopefully, this can reduce reliance on the current mix of daily and weekly regression testing, which is undirected and already consumes several weeks of computer time per week.

Fuzz testing used in SearchSYS gives us automated testing. It consumes computer time, rather than a test engineer's time. Over the last few years fuzzing has been extensively used by Google to find many thousands of security related problems and other bugs [17]. AFL++ [18] is the state of the art fuzzing tool. Our SearchSYS extends AFL++ with the addition of automatically generated test seeds and domain specific mutation operators and, as we shall see, leads to cost effective testing of the state of the art in VLSI simulators, i.e. gem5, for the most widely used general purpose CPU on the planet.

The purpose of this study is to evaluate the reliability and accuracy of gem5 as an ARM simulator and to extend gem5's existing test suite. To achieve this, we apply SearchSYS to identify bugs in gem5, generate test cases, and measure gem5's capabilities on ARM machines.

We identify 14 different types of bugs, including panic errors (which are of particular interest to gem5 developers [19]), performance bugs, and differential bugs, where we compare the simulation outcomes with those from physical ARM machines. We have already reported 4 of these bugs. The selection of which bugs to report depends on whether they belong to the previously mentioned categories. We do not report regular crashes, such as system call issues, non-panic segmentation faults, and timeouts that are less than 24 hours, as the developers find them less interesting [19].

To summarize, our contributions are:

- An extensive empirical study of SearchSYS applied to the ARM architecture, using the gem5 system simulator. Our results reveal that we were able to identify 14 different types of bugs in gem5's simulation of the ARM architecture.
- Delivery of 3 661 LLM-generated tests and 30 000+ fuzzed tests along with a detailed analysis of bug-activating test cases to the gem5 development team.
- An empirical investigation of six large language models for input generation for SearchSYS.

We provide SearchSYS, bug reports, test cases and details of fuzzing performance and statistics via [20], to facilitate reproducibility of this study and wider use of SearchSYS.

Section III describes SearchSYS and how it integrates multiple C code generating LLMs with fuzz testing and extends fuzz testing with simulator specific mutations and differential testing. Our research questions, methodology to answer those, and experimental setup are given in Section IV. Whilst Section V gives our results including contrasting the effectiveness of six LLMs at testing gem5's simulations of ARM hardware. The discussion (Section VI) and related work (Section VII) are followed by our conclusions in Section VIII, but first, we start with the background (Section II).

II. Background

We provide here a quick introduction to fuzz testing and large language models, as these underpin our approach implemented in SearchSYS.

A. Fuzzing with AFL

Fuzzing is a technique used to identify bugs in programs by running the program with a variety of test inputs. Originally the test inputs were generated at random. With the introduction of feedback-based fuzzing techniques, new methods have been developed to manipulate the initial test inputs. Typically these aim to change (mutate) the test inputs in order to exercise new parts of the software under test (SUT) during the fuzzing process. One of the most popular fuzzers is the American Fuzzy Lop (AFL) [21], and related to the AFL family is AFL++ [18], which we are using for SearchSYS.

AFL begins by automatically instrumenting the software under test (SUT). This instrumentation provides feedback to the fuzzing process and is performed at compilation time. The compiler introduces various flags in the programs that are related to the code branches visited during execution. During the fuzzing process, AFL starts by running a set of predefined inputs, called *seeds*. These inputs are then mutated as part of the fuzzing strategy. AFL retains those mutated inputs that explore new sections of the program that previous inputs have not visited, and continues to mutate them to discover new paths. The inputs are maintained in a *queue* and are selected by the fuzzer based on the strategy and the effectiveness of their mutations. The main goal of fuzzing is to improve the test coverage of the SUT. The fuzzer runs until specific termination conditions are met, usually a time limit.

In the context of this work, the seeds are created by large language models (LLMs). The LLMs generate not only programs that exercise the SUT (the simulator), but also the program's inputs and their types.

B. Large Language Models

Like the original SearchSYS work [3], [5], we use large language models to provide a set of seeds for the fuzzers. Large language models have become prominent in recent years, especially after the popular introduction of GPT-3.5. GPT LLMs are based on transformers [22] with attention mechanisms to identify relevant parts during the learning process. Currently, several large language models are available, both private and public. Notable private LLMs include Gemini by Google and Llama 3 by Meta. In terms of public LLMs, various communities have also created their own, such as Dolphin and Mistral, which are available through platforms like Huggingface or the Ollama interface.

Here we use a variety of LLMs of different sizes and natures. We use GPT-3.5 and Phi-2, a Microsoft LLM aimed at software generation. We employ Llama2 and TinyLlama, two general-purpose LLMs of different sizes. We use Magicoder, combining auto-encoders and transformers for program source code generation. Additionally, we use CodeBooga, which integrates various LLMs, specifically Phind-CodeLlama-34B-v2, which outperforms GPT-4 in code generation tasks [23] and WizardCoder-Python-34B-V1.0. All of these public LLMs are run through the Ollama framework, which provides the necessary infrastructure for LLM execution.

C. gem5 System Simulator

gem5[2] is a state-of-the-art discrete time simulator for logic circuits. It is often used to try out the logic design of new electronic components such as memory cache systems, FPGAs, and even CPUs. gem5 is a large open-source project hosted on GitHub, written mostly in C++ and Python. It is used by companies such as ARM and Google to simulate hardware. Including objects, shared library and images, gem5 occupies over 28 GB of memory. It is composed of ~ 1.34 million lines of code, comprising more than a million lines of C++.

The gem5 simulator has a comprehensive testing framework comprising C++ unit tests, Python unit tests, and TestLib integration tests. Each test set focuses on specific aspects of the system, ranging from low-level code validation to full-scale simulation testing.

The **unit tests** in gem5 are designed to validate the functionality of the core C++ components and are automatically executed as part of gem5's continuous integration (CI) process. These test much of the core C++ code and maintain code integrity and correctness after Git commits. In addition to C++ unit tests, gem5 includes Python unit tests, which naturally focus on verifying Python-based components. The Python tests are quick to run and fewer in number than the C++ unit tests.

Testlib integration testing is more intensive in nature. Almost everything in Testlib runs a gem5 simulation. The tests are categorized into three sets: "quick", "long" and "very-long". The "quick" tests are run during CI, the "long" tests are executed nightly, and the "very-long" tests are conducted weekly. Some tests depend on the specific configuration of the host system, and Docker containers are often required to ensure these tests run properly.

[2]https://www.gem5.org

III. SEARCHSYS

Figure 1 shows the structure of each part of SearchSYS. Our approach combines large language models with fuzz testing to identify bugs within system simulators and to generate test suites. The main idea is to address a key challenge with fuzzers: traditionally fuzzers are given a set of input values for the software they are to test (the SUT). (These are known as seeds.) Here we are testing simulators, whose inputs are programs to be simulated. That is, instead of starting with numbers, etc., as inputs, the initial seeds are programs. Until recently, automatically creating programs of interest was hard[3]. However, now there are LLMs dedicated to program source code generation. Therefore, we use LLMs to create an initial set of test seeds, which are of interest for the fuzzing process and can improve test coverage. Given the complexity of the simulator software, it is crucial to have useful seeds that can enhance coverage, especially when testing a comprehensive architecture like ARM. Further, as a traditional fuzzer runs, it changes the SUT's test inputs to try and cover new branches within the SUT. As our SUTs are simulators, SearchSYS extends the fuzzer so that it can mutate both the input executable binary program and its command line parameters along with their data type.

We consider three types of bugs: 1) crashes 2) issues with efficiency or performance, such as program hangs 3) mismatches between the simulator and the actual hardware. For this last type of bug, it is essential to determine whether the simulator behaves as the real system. This is where differential testing becomes crucial. SearchSYS provides the differential testing infrastructure to compare test cases within the simulator and outside of it.

Our testing approach is divided into three main components:

1) **Test Input Generation System**: This system uses Large Language Models (LLMs) to create a set of test cases which serve as our testing baseline. Unlike traditional fuzzing processes that require existing programs as seeds, we ask different large language models to generate the test cases (programs to be simulated) automatically. (Note: in the context of the ARM Instruction Set Architecture it can take weeks for a skilled engineer to sit down with the ARM ISA hardware documentation and code programs to test each part of the ISA.)
2) **Coverage-Guided Fuzz Testing Tool (AFL++)**: AFL++ focuses on identifying new uncovered code regions in the simulation system and achieving higher coverage through mutation-based testing. AFL++ employs a driver for the simulation system, enabling the simulation/execution of any binary program with specific parameters. The type of the parameters is specified by the LLM. The LLM

[3]E.g. (in the context of testing) automatically generating programs that achieve high coverage, have specific patterns or formats, or contain edge cases or a specific set of instructions.

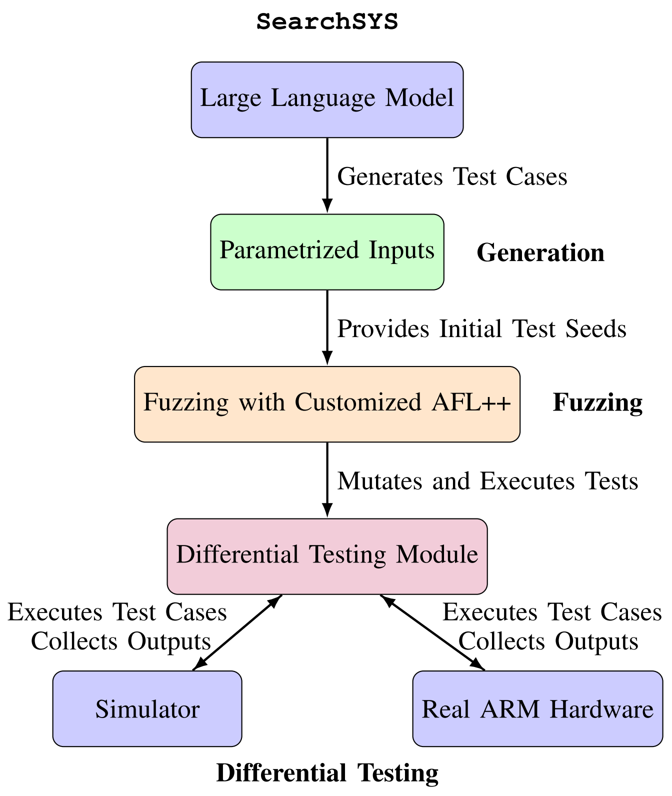

Fig. 1. `SearchSYS` diagram starts with the generation phase, where the LLMs generate C programs which take parameters via the command line. It compiles them. Then it applies the fuzzing phase, using the customized version of AFL, which is able to mutate both the executable binary and its parameters. Finally, in the differential testing phase, it compares each mutated executable binary in both the simulator and the real ARM architecture.

not only provides the program to be simulated but also specific inputs for it, which our special version of `AFL++` search will modify along with the program itself.

3) **Differential Testing Module**: This module compares the outputs of the simulator with those of the real ARM architecture. We run the specific binary files and their inputs on a real ARM machine and check if the outputs from it and the simulator match. Any discrepancies are identified as mismatches.

A. LLM-based test code generation

The first step of `SearchSYS` focuses on creating a corpus of test cases with parameterized inputs. To achieve this, we query a large language model (LLM). The LLM generates the test cases as C source code and provides the types for the input parameters. The query prompt given to the LLM specifies the tasks the program should achieve and also requests it to provide the corresponding parameters and their types.

We employ a zero-shot prompting technique [24]. That is, the LLM is not given any prior examples or context. Hence we have to provide all the information the LLM needs about the specifics of the problem in its query prompt (see shaded example in Listing 1). We construct a prompt to generate a new test program, repeated n times to produce n programs, where n can be arbitrarily large. In total, $n + 1$ prompts generate n test programs. One prompt for setting the LLM's role plus n prompts to generate n test programs.

a) Setting the LLM role: Before generating the n test programs, a single one-off LLM prompt initializes the process. This one-off LLM prompt serves to set the LLM's role. This LLM prompt specifies the programs to be generated (e.g. "generate C programs with arguments as input"). Since each LLM has memory [4], we only need to set LLM's role once.

b) Generating a test program: The LLM prompt starts with a problem description that specifies the program tasks and the context in which these programs are to be generated. The problem description contains specific gaps that will be filled with the tokens. A total of 4089 tokens are available as part of `SearchSYS`. Table I describes the three token types across 4 categories of tokens with a few examples. The first and second types of tokens are derived from compiler optimization names and compiler parts. The third token is chosen at random from general tokens related to programming languages, tutorials, or standards. The fourth tokens category in Table I includes phrases taken from the C17 standard [25], such as "initialize" and "pointer". For instance, we might ask for examples of dead code elimination, handling of the Abstract Syntax Tree (AST), and a beginner tutorial name in C programming examples, corresponding to the first, second, and third tokens.

The template prompt is shown in Listing 1.

```
"Coding task: give me a program in C with all
includes. Input is taken via argv only.
Please return a program (C program) and a concrete
example of an input (BASH). The C program will be
with code triggering " + <Token-1> + "optimisations,
covers this part of the compiler " + <Token-2> + ",
and exercises this idea in C: " + <Token-3> + ".
To recap the code contains these:" + <Token-1> +
" and " + <Token-2> + " and " + <Token-3>;
```

Listing 1. Template prompt for generating C test programs with random tokens

The value of the `<Token-n>` comes from the three different components and is chosen at random from within each corpus. This process fully automates the fuzzer's input generation and improves the diversity of the test inputs. Additionally, it is independent of the specific LLM being used.

B. Fuzzing

`SearchSYS`'s fuzzing process mutates both the binary program and its inputs. When employing bit-flip mutation, as standard in `AFL++`, context consideration is crucial. Mutations causing binaries to fail to load or execute even a single instruction lead to inefficient SUT testing, reducing the likelihood that developers will invest time in identifying or fixing bugs. For instance, binaries failing with errors like `"error while loading shared libraries: unsupported version 0 of Verneed record"` due to bit-flips are unlikely to receive fixes from developers since the correct behaviour for faulty binaries is to crash. To

[4]i.e. the LLM's consistency within a session.

TABLE I
THE FOUR C TOKEN CATEGORIES WHICH ARE USED TO FILL GAPS IN THE TEMPLATE USED TO PROMPT THE LLMS AND A FEW EXAMPLES OF EACH

ID	Type	Category	#Tokens	Examples
1	Token-1	Compiler Optimizations	26	"Scalar Optimizations", "Dead Code Elimination", "Constant Folding"
2	Token-2	Compiler Parts	36	"Sema", "Serialization", "Parse", "Lex", "AST"
3	Token-3	Domain Problems	197	"C Program to Sort an Array using Merge Sort", "Calendar Year in Different Formats", "input includes several arguments"
4	Token-3	Compiler Standard Indices [25]	3830	"AND operators", "cimagl function", "EOF", "locale", "pow", "SCNiMAX"

address this, `SearchSYS` employs a custom bit-flip mutation operator that controls the number and frequency of bit-flips, applying them only to a program's compiled binary file. This avoids applying bit-flips to arguments or type information, preserving the structure of the test input.

`SearchSYS` applies three mutation operators for testing system simulators:

1) A bit-flip operator for modifying a program's compiled binary file.

2) A range-enhanced operator for editing argument values within their specified type range.

3) An operator for changing the value's type.

Operator (2) uses type information to ensure arguments remain valid, while Operator (3) randomly changes the type, such as from `INT32` to `LONG`, potentially exposing memory safety issues in the SUT. We support all integer types, float, double, and strings but have not yet implemented pointer support. Figure 2 and Figure 3 provide examples of value (operator 2) and type mutations (operator 3), respectively.

```
./mutator_args.so test.o, 5:INT, 20:LONG , "Hi":
    STRING
After Mutation: test.o, 10:INT, 20:LONG, "House":
    STRING
```

Fig. 2. Example of mutation operator (2) changing argument values: first argument: 5 to 10 and third: "Hi" to "House" (second argument unchanged).

```
./mutator_args.so test.o, 5:INT, 20:LONG , "Hi":
    STRING
After Mutation: test.o, 5:LONG, "20":STRING, "Hi":
    STRING
```

Fig. 3. Example of mutation operator (3) changing argument types: first argument from INT to LONG and second argument from LONG to STRING.

`SearchSYS` loads all three mutators (1-3) using the existing `AFL++` option, allowing `AFL++`'s heuristics to select the next mutation operator. However, we decrease the probability of choosing (2) by setting it to only 99.5% of the times `AFL++` selected it originally and replacing the remaining 0.5% with mutator (1), as `AFL++` favours this operator due to its low failure risk, which is too conservative for fuzzing.

`SearchSYS` extends `AFL++` by evaluating new test inputs in the form of `binary name, arguments list, types`. Then it applies mutation operators directly to the compiled binaries, their arguments, or their arguments' types. This approach allows for more complex mutations, such as binary file mutations.

Another factor affecting the throughput of fuzzing is the number of mutation operations that `AFL++` performs in a single iteration. The parameter `afl_custom_fuzz_count` controls the number of times a test input should be mutated and executed against the target. A lower value reduces the risk of iteration failure but can lead to inefficient fuzzed input generation due to iteration overhead. Following our results [5], we set `afl_custom_fuzz_count` to be `17`, `84` and `66` for Operator 1, 2 and 3 from Section III-B, respectively.

C. Differential Testing

Fuzzed test inputs can uncover crashes, hangs and mismatches between the architecture and the system simulator. However, `AFL++` only identifies a bug if the test input leads to a crash or a hang, degrading `SearchSYS`'s ability to detect missimulation issues. To address this, `SearchSYS` follows the approach of Even-Mendoza et al. [26] and separates fuzzing and differential testing. That is, after fuzzing we perform differential testing by comparing runs using fuzzed test inputs (i.e. mutated executable binaries and their inputs) on ARM hardware (native) with those on the simulator with ARM ISA.

D. Implementation

`SearchSYS` is implemented using a mixture of languages and Unix shell scripts. Specifically, we use (1) Java with Ollama for LLM-based test input generation, (2) C/C++ for the custom mutator and fuzzing-related code, and (3) a set of scripts for configuring the testing environment [3], [5].

When adapting `SearchSYS` for ARM, we encountered minor script issues related to the linker and data from `TinyLlama`, the latter caused an early termination in `AFL++` fuzzing due to memory errors. Additionally, we had to pass ARM as the ISA parameter, though no modifications were needed for the Python configuration script. We continued to use the example script provided by the SSBSE Challenge Track 2023 organizers[5]. These adjustments required minor modifications to our scripts (setting the testing environment, fuzzing and differential testing parts).

The code in parts (1) + (2), required no further modifications. However, in between this version and [5], we made some bug-fixing edits in the C/C++ code.

[5]`hello-custom-binary.py`

IV. Evaluation

To assess the quality of `gem5` as a simulator for ARM architectures, we aim to answer the following questions:

RQ1: Considering that large language models (LLMs) generate test suites independently, how effective are these test suites at identifying bugs in `gem5`'s ARM simulation processes?

To answer this question, we will ask LLMs to generate test suites, which will later be used as seeds for the fuzzing process. Each test program (of a test input in the suite) will be generated using a different prompt: `SearchSYS` generates random tokens (Section III-A), embeds the random tokens into the template prompt in Listing 1 to form a complete prompt, and prompts the LLM with the constructed prompt to generate a new test program. These steps are part of `SearchSYS`'s test input generation system (Section III).

This research question will help determine which specific language models are more effective at generating test suites that can uncover bugs in the simulator, under the current prompting mechanism (Section III-A). In this evaluation, we will identify the best language model for finding bugs in the ARM simulation process and uncover specific bugs. Our process will also ensure a minimum set of test cases after reducing the test suite using afl-cmin[6].

RQ2: How effective is `SearchSYS` at identifying bugs in the simulation of the ARM architecture performed with `gem5` after the fuzzing process?

To answer this research question, we will run the fuzzing strategy using the different test suites generated by each LLM as seeds. The fuzzer will run against the simulator, with the generated inputs focusing not only on identifying crashes and hangs within the simulator but also on mismatches in differential testing by comparing the output of the generated tests with that of a real ARM machine.

A. Hardware

We ran tests on `gem5` using two machines: **(1)** a single CloudLab[7] [27] m400 machine with 64 GB RAM, ARMv8 64-bit architecture with a single socket, 2.4 GHz, 8 cores, and 1 thread per core, running Ubuntu 22.04 ARM, and **(2)** a single machine (UCL) with 224 CPU cores (Cavium ThunderX2 CN9975, 2.0 GHz) and 130 GB RAM, ARMv8 64-bit architecture with 2 sockets, 28 cores per socket, and 4 threads per core, running Red Hat Linux (aarch64-redhat-linux-gnu). We installed the same compiler versions, adapted for ARM, and set up the same tools as described in [5], on both machines. The exact specification can be found in [28].

B. Experiments

For the experiments, we selected ARMv8 hardware and the ARM ISA, given its relevance from the industry's perspective [19]. The primary objective was to provide a comprehensive evaluation of `SearchSYS`'s performance across different architectures. To achieve this, we compared the bug-finding effectiveness and fuzzing throughput of `SearchSYS` between the ARM ISA and X86 ISA, as previously explored in [5]. We followed the experimental procedure outlined in [5], but adapted it for the ARM ISA and ARMv8 hardware, running without Docker. Furthermore, the experiments were repeated 5 times (instead of 10), using the minimized input corpora, as these have been shown to be more stable during fuzzing and in particular for `SearchSYS` [5], [29], [30].

The minimized input corpora `TinyLlama`, `Phi2`, `Llama2`, `Magicoder`, `CodeBooga` and `GPT-3.5-turbo` were taken from [5], [6], while `GPT-3.5-turbo` (SSBSE 2023) from [3], [4].

Each fuzzing campaign ran for 24 hours, with five independent repetitions of the fuzzing process per minimized input corpus. The throughput results of the fuzzing experiments were calculated as the mean value across these five repetitions, ensuring consistency and robustness in the findings. The differential testing post-fuzzing (i.e. when `gem5` differed from the real hardware) was done using the last repetition, for all 7 corpora. In case of a mismatch, to determine if the mismatch is a genuine bug, we compared the results from the `gem5` simulation with ARM ISA to those obtained via the two ARM hardware (Section IV-A).

V. Results

In this section, we present the outcomes of our experiments, specifically, bugs found from the initial corpus of LLM-generated test inputs, and those found as a result of fuzzing campaigns. To detect bugs, we cross-validated our results between an X86 and two ARMv8 machines. One of these ARMv8 machines is being configured to be more strict, often initializing uninitialized local variables to zero.

In total, we reported 4 bugs from the LLM-generated test inputs [31]–[34]. We further identified a bug in the GNU Multiple Precision Arithmetic Library (GMP 6.1.0) during cross-system testing [35].

A. Bugs Detected from LLM-generated Test Inputs

To evaluate LLMs as sources of inputs for regression testing of system simulators, we used 7 sets of LLM-generated test inputs, created during 25-hour runs with different LLMs with qualitative and quantitative evaluation and analysis of LLM-generated test inputs detailed in [3], [5]. Here, we focus on ARMv8, noting that LLM test program generation is agnostic to X86 or ARM CPU. We classified the bugs found for each of the 7 sets of simulations with ARM ISA against real ARMv8 machines and compared the bug finding rate with our previous results on the `gem5` X86 backend [5].

Table II shows the bugs identified in our investigation from the LLM-generated test inputs. Table II includes the bug description ("Bug") with a bug number if already reported to the `gem5` bug tracking system. Columns A to G represent the number of test inputs triggering this bug category: **A**: `TinyLlama`, **B**: `Phi2`, **C**: `Llama2`, **D**: `Magicoder`, **E**: `CodeBooga`, **F**: `GPT-3.5-turbo` and

[6]afl-cmin and afl-cmin's manpage

[7]See https://docs.cloudlab.us/hardware.html

TABLE II
BUGS FOUND IN gem5 WITH LLM-GENERATED TEST INPUTS. COLUMNS A TO G HOLD THE NUMBER OF INSTANCES OF EACH BUG FOUND BY EACH LLM: **A**: TINYLLAMA, **B**: PHI2, **C**: LLAMA2, **D**: MAGICODER, **E**: CODEBOOGA, **F**: GPT-3.5-TURBO AND **G**: GPT-3.5-TURBO (SSBSE 2023).

BUG	**A**	**B**	**C**	**D**	**E**	**F**	**G**
Bug #1527 (panic)	**1**	**1**	**5**	**1**	**1**	0	0
Bug #1544 (Missing support)	0	0	**2**	**3**	**6**	**5**	0
Bug #1547 (Missing support)	0	0	0	**1**	**7**	0	0
src/sim/syscall_emul.cc:67: fatal: syscall dup3 (#24) unimplemented	0	0	0	**3**	0	**14**	0
src/sim/syscall_emul.cc:67: fatal: syscall pipe2 (#59) unimplemented	0	0	0	**1**	**1**	0	0
src/sim/syscall_emul.cc:67: fatal: syscall clock_getres (#114) unimplemented	0	0	0	0	0	**1**	0
src/sim/syscall_emul.cc:67: fatal: syscall clock_nanosleep (#115) unimplemented	0	**1**	**8**	**7**	**4**	**4**	0
src/sim/syscall_emul.cc:67: fatal: syscall wait4 (#260) unimplemented	0	0	**2**	0	0	0	0
instruction 'bti' unimplemented	**20**	**17**	**81**	**127**	**51**	**202**	**4**
Bug #1629 (some `time` functionality unimplemented)	0	0	0	**2**	**1**	**28**	0
some `thread` functionality unimplemented in SE mode	0	0	0	**1**	0	**10**	0
Variable's value is random in ARM but fixed in simulation	0	0	0	**2**	**1**	**7**	0
Timeout	**1**	**2**	**4**	**27**	**15**	**15**	**1**
Totals (of number of test inputs exposing a bug)	22	21	102	175	87	286	5

G: GPT-3.5-turbo (SSBSE 2023). The bugs found include panic errors, usually triggered by assertion violation, timeouts, and different outputs including fatal errors triggered by wrong or missing instruction implementation for ARM ISA, revealing gaps in the system's support for ARMv8 functionalities. We ran the differential testing scripts comparing the result of the native run on ARMv8 machines against the simulation with a time out of 50 seconds and a memory limit of approximately 90 MB (stack size) for both the gem5 simulation and the native run.

In total, GPT-3.5-turbo found the highest number of issues (286), followed by Magicoder (175), Llama2 (102), CodeBooga (87), TinyLlama (22), and Phi2 (21). GPT-3.5-turbo (SSBSE 2023) had the fewest, with only 5 instances. Two of these bugs, panic and timeout, were identified by executing the simulator, totalling 74 test cases (9 panic and 65 timeouts), while an additional 624 bug-activating test cases were identified using differential testing.

The differing results for GPT-3.5-turbo (SSBSE 2023) compared to the GPT-3.5-turbo set, even though both are using the same language model, can be attributed to three factors: (1) GPT-3.5-turbo (SSBSE 2023) dataset was generated in 2023, while GPT-3.5-turbo dataset was generated in 2024, (2) GPT-3.5-turbo (SSBSE 2023) was trained on the LLVM test suite, likely overlapping with tests already evaluated by gem5, and (3) GPT-3.5-turbo (SSBSE 2023) used a few-shots approach instead of zero-shots, with zero-shots generally providing better input diversity and throughput [5]. We observed these differences for the bug count results during the pre-fuzzing test input generation stage (Table II), but as well in the results from the fuzzing and post-fuzzing stages (to be discussed in Section V-B).

These LLM test cases are a suitable contribution to gem5's C++ unit tests (as regression tests). They have several lines of code, are efficient (as they commonly terminate in under 50 seconds) and are human-readable. With some semi-manual filtering, to remove tests triggering undefined behaviour or those expecting complex input or having large output, these can be grouped according to coverage and instructions triggered to target untested areas of the current regression tests of the gem5 test suite. Furthermore, test inputs exposing unimplemented features can be saved for future use. For example, tests triggering unimplemented system calls can be saved for future use (e.g. to implement test-case-driven development methods).

In most cases, unimplemented syscalls represent a gap in gem5 functionality which the gem5 developers do not intend to plug, but the bug report itself is helpful since it can alert gem5 users to a now-known issue, saving them time trying to resolve what was previously an issue known only to the developers.

In gem5 using SearchSYS, we found around 530 test inputs exposing missimulations and errors and 30 test inputs exposing optimization issues (timeouts) on X86 ISA [5] compared to around 630 and over 60 instances on ARM. Note, that the '530' and '30' instances on X86 are not a subset of the 630 and 60 instances on ARM and reflect different bugs. This suggests that contrary to expectations [19], ARM is no more stable than X86 ISA in gem5, as given the same set of test inputs, more of them exposed an issue.

We reported four new bugs from the LLM-generated test inputs to the gem5 bug tracker [31]–[34] under "arch-arm" tag. These bugs were not previously identified during our X86 fuzzing campaigns [3], [5], except for bug #1629 [34], which we expected to be properly implemented in ARM. While we anticipated that ARM would be generally more stable than X86, the occurrence of this bug in ARM was unexpected, prompting us to report it immediately. We excluded unimplemented issues from bug reports, as these represent new test cases rather than valid bugs.

RQ1 Answer. All of the LLMs found bugs in gem5's simulation of ARM CPUs. We found 13 bugs, six are unimplemented system calls functionality in the simulator. We have already reported four (#1527 #1544 #1547 #1629). Of these three have already received detailed consideration by the gem5 development team.

B. *Fuzzing as Part of the Testing Process*

SearchSYS is an AFL++-based tool. After generating test inputs using LLMs and compiling them into a test case (binary, input, and input type info), we fuzzed an instrumented version of gem5 for 24 hours to have sufficient time to explore the codebase of a specific gem5 version [36].

Table III shows throughput during 24 hours of fuzzing, detailing the number of new test inputs generated for each of the 7 minimized input corpora (from each of the LLMs used). The columns "Initial Corpus" indicates the size of each initial corpus at the start of the fuzzing, "Fuzzed Corpus (Std Dev)", "Queue (Std Dev)", "Crash (Std Dev)" and "Hangs (Std Dev)" are means over five trials per corpus with their observed standard deviations. The "Fuzzed Corpus" column shows the total number of test inputs generated (i.e. queue + crashes + hangs). At the end of each 24-hour fuzzing campaign, we recorded: (1) "Queue" (test inputs generated by AFL++ that did not crash or hang and were therefore suitable for further mutation), (2) "Crash" (number of crashed test inputs), and (3) "Hangs" (number of hanged test inputs).

The minimized input corpora vary in size, with approximately 100 test inputs generally being recommended [29], [30]. During fuzzing, the highest throughput (in total) of fuzzed test inputs was achieved (on average) by GPT-3.5-turbo (SSBSE 2023) with 997 fuzzed test inputs, followed by Magicoder (986), Llama2 (948), GPT-3.5-turbo (888) and CodeBooga (839). The smaller LLMs had a lower throughput during fuzzing: TinyLlama (776) and Phi2 (680). This trend was slightly different when observing the queue size only (the fuzzed test inputs for differential testing post fuzzing), with Magicoder achieving the highest rate with 962 fuzzed test inputs, followed by GPT-3.5-turbo (SSBSE 2023) with 945 fuzzed test inputs. This is different from the pattern we already reported in [5] for X86, where GPT-3.5-turbo (SSBSE 2023) and TinyLlama achieved the best throughput while GPT-3.5-turbo had extremely poor fuzzing throughput. Lastly, CodeBooga showed a large standard deviation across all measured outputs (Queue, Crash, and Hangs), while Llama2, Phi2, and TinyLlama had high deviations in some outputs, though smaller than CodeBooga's. The remaining LLMs exhibited generally smaller standard deviations. Despite these variations, the minimized input corpora typically generated between 700 and 1,000 test inputs during 24 hours of fuzzing.

Table IV: shows the bugs identified by comparing when gem5 simulated the mutated binary programs with running them on the real hardware. Table IV includes the bug description ("Bug") with a bug number if already reported to the gem5 bug tracking system. Columns A to G represent the number of test inputs triggering this bug category: **A**: TinyLlama, **B**: Phi2, **C**: Llama2, **D**: Magicoder, **E**: CodeBooga, **F**: GPT-3.5-turbo and **G**: GPT-3.5-turbo (SSBSE 2023). Note "Crash" and "Hangs" from Table III: (1) Table IV does not include timed-out fuzzed test inputs counters since these are already stated in Table III in the "Hangs" column; and (2) not all instances counted in the "Crash" column of Table III represent genuine crashes; some are due to corrupted binaries or non-reproducible crashes. Since AFL++ categorizes all these as a single "Crash" type, we included these in Table IV and refined it for a more detailed breakdown.

Table IV identifies hangs and 7 distinct issues discovered during a manual inspection of automatically flagged mismatches, warnings, and crashes. During 24-hour fuzzing, TinyLlama had the highest number of fuzzed test inputs exposing issues (87), followed by GPT-3.5-turbo (31), Magicoder (10), Phi2 and Llama2 (6 each), and CodeBooga (3). Most fuzzed sets identified 2-3 distinct issues, with GPT-3.5-turbo finding 4 distinct ones. As in Table II, GPT-3.5-turbo (SSBSE 2023) had the lowest bug-finding rate with only 2 instances, each of a different issue category.

Some previously known bugs were encountered during fuzzing, generating further examples of the issue, which can be useful for bug localization and debugging. We found one "Out of Memory" instance. This indicates that fuzzing can be beneficial, though it suggests that longer fuzzing runs may be necessary for uncovering additional bugs and a better understanding of the codebase code coverage. During fuzzing, we also identified a fatal error in src/mem/port_proxy.hh readBlob, which at first, looked like a genuine error. However, comparing the results between the two ARMv8 machines, with the simulation failing only on the UCL machine (Section IV-A) led us to conclude that it is likely to be a configuration issue rather than an ARM ISA bug, and thus we excluded these from the tables.

To recap, during the fuzzing campaign, three of these bugs, panic, out-of-memory and timeout, were identified by executing the simulator, totalling 93 test cases (19 panic, 1 out-of-memory and 73 timeouts), while an additional 126 bug-activating test cases were identified using differential testing.

Fuzzing is a time-intensive process, and as such is unsuitable for regular runs of regression tests, where the developer requires feedback quickly. However, due to its ability to produce diverse inputs covering unexpected branches, it could be valuable for use in less common, but large test suites, such as those run between new releases, where there is a larger time budget. Fuzzing can then help identify obscure bugs which may not have been caught during the other stages of development. In the context of gem5, fuzzing of weekly or release versions can be integrated into the TestLib process of gem5, which can include using Docker containers to provide

TABLE III
NUMBER OF gem5 TEST INPUTS GENERATED BY EACH INITIAL CORPUS (BY LLM, MEAN AND STANDARD DEVIATION OF 5 RUNS). SEE SECTION V-B

	Initial Corpus	**Fuzzed Corpus (Std Dev)**	**Queue (Std Dev)**	**Crash (Std Dev)**	**Hangs (Std Dev)**
`TinyLlama`	206	776 (±59)	737 (±59)	29 (±4)	10 (±1)
`Phi2`	366	680 (±87)	651 (±82)	14 (±6)	15 (±3)
`Llama2`	613	948 (±62)	916 (±62)	16 (±6)	16 (±3)
`Magicoder`	719	986 (±41)	962 (±42)	13 (±2)	11 (±3)
`CodeBooga`	612	839 (±112)	816 (±106)	11 (±7)	12 (±6)
`GPT-3.5-turbo`	703	888 (±26)	871 (±24)	11 (±3)	6 (±3)
`GPT-3.5-turbo` (SSBSE 2023)	442	997 (±34)	945 (±34)	49 (±5)	3 (±3)

TABLE IV
DIFFERENTIAL TESTING RESULTS. BUGS FOUND IN gem5 DURING 24-HOUR FUZZING OF THE LAST REPETITION. COLUMNS A TO G HOLD THE NUMBER OF INSTANCES OF EACH BUG FOUND BY INITIAL MINIMIZED INPUT CORPUS: **A**: TINYLLAMA, **B**: PHI2, **C**: LLAMA2, **D**: MAGICODER, **E**: CODEBOOGA, **F**: GPT-3.5-TURBO AND **G**: GPT-3.5-TURBO (SSBSE 2023). (SEE SECTION V-A FOR EXPLANATION OF DIFFERENCES BETWEEN **F** AND **G**.)

BUG	**A**	**B**	**C**	**D**	**E**	**F**	**G**
src/sim/syscall_emul.cc:67: fatal: syscall dup3 (#24) unimplemented	0	0	0	0	0	**4**	0
src/sim/syscall_emul.cc:67: fatal: syscall pipe2 (#59) unimplemented	0	0	0	0	**1**	0	0
src/sim/syscall_emul.cc:67: fatal: syscall clock_nanosleep (#115) unimplemented	0	0	0	**4**	0	**7**	0
instruction 'bti' unimplemented	**21**	**3**	**3**	**5**	**1**	**19**	**1**
Out of Memory	0	0	0	0	**1**	0	0
Likely to be Bug #1544 (Missing support)	**56**	0	0	0	0	0	0
Bug #1527 (panic)	**10**	**3**	**3**	**1**	0	**1**	**1**
Totals (of number of test inputs exposing a bug)	87	6	6	10	3	31	2

a consistent environment for fuzzing as in [37], ensuring reproducibility and isolating system dependencies. Furthermore, `SearchSYS`, a coverage-directed fuzzer, can assist in exploring newly added code by using partial instrumentation or tailoring the mutators, enabling deeper testing of the codebase of `gem5`.

Our fuzzing campaigns have uncovered further bugs and produced a larger corpus of test cases than LLM-generated tests alone. We found crashes and hangs using all LLMs tested. Even with small LLMs such as `TinyLlama`, fuzzing was able to grow the corpus size from 206 input tests to 776 input tests, including finding 29 crashes, and 10 hangs.

RQ2 Answer. `SearchSYS` generated 30 000+ test cases for `gem5`. The majority of bugs, except for panic errors, out-of-memory and timeouts, were identified using our differential testing mechanism. `gem5` developers' feedback reveals that `SearchSYS`'s ability to tie test cases to particular features of the ARM ISA is of great importance, and could help with ongoing development of `gem5` also for RISC V and other ISAs.

VI. DISCUSSION

The use of LLM-generated test inputs has proven to be a valuable method for uncovering new bugs within the `gem5` system, particularly when evaluating the ARMv8 ISA. During our experiments, 7 distinct sets of LLM-generated inputs were used, and the bugs identified were systematically categorized based on their occurrences in simulations. Notably, we discovered a range of panic errors, timeouts, and fatal errors, predominantly triggered by assertion violations or improper/missing instruction implementations for ARM ISA. This highlights significant gaps in `gem5`'s support for ARMv8 functionalities.

Our comparative analysis revealed that ARM is not necessarily more stable than the X86 ISA within `gem5`, contradicting previous expectations [19]. This higher incidence of issues on ARM suggests that there are still considerable challenges to achieving parity between the support for ARM and X86 within `gem5`.

The efficiency and human readability of the LLM-generated test cases make them excellent candidates for integration into `gem5`'s C++ unit tests as regression tests. They run quickly and we can exclude tests that trigger undefined behaviour, require specific input, or produce large outputs. This allows for targeted testing of untested areas, improving the overall robustness of the `gem5` test suite. In addition, tests that reveal unimplemented features can be earmarked for future development, supporting a case-driven test development approach.

As mentioned in the next section, recently LLMs have proved very popular in software engineering research and their rapid development means any paper will lack recency, nonetheless our selection of LLMs provides good coverage. The analysis in [5] required approximately 3500 hours of machine time. There is no need to repeat the LLM-based test program generation as: (1) by the time the analysis is completed, newer LLM versions would render it outdated, and

(2) it is CPU-agnostic[8], relying on GPU or OpenAI platform.

Our fuzzing campaigns significantly expanded the corpus size and identified numerous crashes and hangs across all LLMs. For example, even with a smaller LLM such as TinyLlama, the initial corpus size grew considerably, uncovering several crashes and hangs. This demonstrates the ability of fuzzing to produce diverse inputs that explore unexpected branches, making it a valuable complement to other testing methods.

For corpus seed generation, the primary objective was achieving a high compilation rate for effective test program generation, leaving test input diversification to fuzzing. The compilation rate is critical [26], [38], for two fundamental reasons: (1) in grey-box fuzzing, the priority is on seeds enabling the fuzzer to explore less-covered areas effectively, especially when the diversity metric relevant to the SUT is unknown to the LLM, and (2) diverse but failed to compile code is useless, as it yields no executable for simulation. This objective is unlikely to be fully achieved with prompt engineering alone, and we leave this to future work. Nonetheless, frameworks like ROCODE [39], which achieve higher compilation rates via backtracking and code analysis, might be used with SearchSYS to enhance its bug-finding capabilities.

SearchSYS implements a differential testing approach to uncover missimulation bugs (Figure 1). Hence, it requires access to physical hardware matching the simulated architecture, as the correct output of the test inputs may be difficult to determine. This challenge is commonly known as the oracle problem [40]. We acknowledge that access to diverse physical hardware may not always be feasible. In such cases, alternative strategies may be explored, such as leveraging similar architectures or replacing the hardware with another different simulator.

VII. Related Work

There has been extensive interest in applying Large Language Models to Software Engineering. For example, Hou et al. [41] mention seven existing surveys (e.g. [42]) before their own, which covers 2017 to the start of 2024 and includes 395 papers. However, neither Wang et al. [42] or the earlier Fan et al. [43], nor the most recent Hou et al. [41] or [44] include work directly on fuzz testing of simulators.

LLMs have already made an impact in industry (e.g. work in Meta reported by Alshahwan et al. [16]). Other works include [45], [46] where Feldt et al. argue for the use of testing to kerb LLMs' enthusiasm to make up answers. Also missing from both surveys is Le et al. [47], where they propose using LLMs to assist testing of software following the RESTful web API architecture.

ChatFuzz (Hu et al. [48]) aims to improve fuzz testing. Like our SearchSYS it uses an LLM to automatically generate seeds for fuzz testing which are similar to existing seeds but fit better to a given format. Where the format was simple Hu et al. [48] show it can do better than vanilla AFL++. However, they test only one LLM (ChatGPT), and test ChatFuzz on three real-world benchmarks which require structured inputs rather than an open-source simulator used by industry whose inputs are executable programs.

Our approach leverages AFL++ (Section II-A), specializing its application to mutate binary files and their inputs. Even if this approach is uncommon, similar approaches can be found in compiler testing [49], [50] and fuzzing network protocol analysis [51],

Simulation tools like gem5 provide rich information for evaluating architectures and have been widely used for different aspects of the ARM architectures [52]–[54]. Testing complex systems, such as gem5, involves not only efforts to verify its architectural compliance [55] and code integrity but also deeper internal testing or verification methods, such as SearchSYS, which are crucial for validating its numerous options [56]. Similarly Xia et al.'s Fuzz4All [38] also uses LLM to example code snippets for fuzz tester input. Although they consider differential testing in fact in [38] they used traditional "fuzzing oracles, such as crashes".

Serebryany et al. [57] discuss a hardware fault diagnosis tool employed in Google data centres, where test cases were generated through software fuzzing of CPU simulators akin to gem5. They highlight gem5 as a potential tool for future use. On the other hand, Rajeev et al. [58] used gem5 to test their fuzz inputs rather than fuzzing gem5 itself.

Our previous work [3], [5] stands out as the only instance which has been using fuzzing to test gem5 so far. Initially, we integrated LLMs and SBSE to test system simulators and developed a prototype of SearchSYS. With SearchSYS [5], we fully automate the process and enhance the tool with new mutation operators. Specifically, we advance the previous work by having LLM generate an example program instead of providing one ourselves, and by constructing independent mutators to give automated feedback to AFL++ for mutation selection. In addition, in this work, we show how this approach supports testing the ARM architecture by also identifying deep vulnerabilities in these kinds of systems.

VIII. Conclusions

In this work, we show how effective SearchSYS is at generating tests for the gem5 software simulator. In particular, we focus on the simulation of the ARM silicon chip Instruction Set Architecture (ISA). SearchSYS uses LLM-generated programs as seeds in a fuzzing campaign with specialized mutation operators to generate test cases. Through differential testing, we identified 624 bugs with LLM-generated test cases and 126 with fuzzed test inputs. Out of the total number of bug-activating test cases, 4 unique bugs have been reported and acknowledged by developers. In 24 hours of compute time SearchSYS typically generates about a thousand test cases with their line coverage. Taking specifically the problem of testing gem5's implementation of the ARM instruction set architecture, it would have taken a skilled engineer weeks to generate test cases starting from the ARM ISA documentation.

[8]Yet, we adjusted the dataset [5] by re-compiling the test input binaries for the ARMv8 target.

Acknowledgment

We thank CloudLab [27] for the platform and infrastructure support that enabled the experiment resulting in Table II and reproducibility check of bugs when we classified them. This work was supported by the UKRI TAS Hub grants no. EP/V00784X/1 and EP/V026801/2 and the Alan Turing grant G2027 - MuSE.

References

[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, aug 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[2] B. R. Bruce, A. Akram, H. Nguyen, K. Roarty, M. Samani, M. Fariborz, T. Reddy, M. D. Sinclair, and J. Lowe-Power, "Enabling reproducible and agile full-system simulation," in *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*. Stony Brook, NY, USA: , March 28-30 2021, pp. 183–193. [Online]. Available: https://doi.org/10.1109/ISPASS51385.2021.00035

[3] A. Dakhama, K. Even-Mendoza, W. B. Langdon, H. D. Menéndez, and J. Petke, "Searchgem5: Towards reliable gem5 with search based software testing and large language models," in *SSBSE*. Springer, 2023, pp. 160–166, best challenge track paper. [Online]. Available: https://doi.org/10.1007/978-3-031-48796-5_14

[4] ——, "Artifact of SearchGEM5: Towards reliable gem5 with search based software testing and large language models," Sep. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8316685

[5] ——, "Enhancing search-based testing with LLMs for finding bugs in system simulators," Research Square, 18 September 2024. [Online]. Available: https://doi.org/10.21203/rs.3.rs-5004178/v1

[6] ——, "Artifact of enhancing search-based testing with LLM for finding bugs in system simulators," Aug. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.13450472

[7] R. Holway, "Monday 30 September 2024 Intel and ARM," 30 September 2024, retrieved 7 Oct 2024. [Online]. Available: https://www.techmarketview.com/ukhotviews/archive/2024/09/30/intel-and-arm

[8] S. Segars, "Arm partners have shipped 200 billion chips," Blog, 18 Oct 2021, retrieved 7 Oct 2024. [Online]. Available: https://newsroom.arm.com/blog/200bn-arm-chips

[9] Arm Education, "Arm research starter kit on system modeling using gem5," https://github.com/arm-university/arm-gem5-rsk, 2024, accessed: 10-Oct-2024.

[10] gem5 developers, "Extending gem5 for ARM," https://www.gem5.org/documentation/learning_gem5/part1/extending_configs, 2024, accessed: 10-Oct-2024.

[11] M. Fowler, "Continuous integration," Blog, 18 January 2024. [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html

[12] T. E. J. Vos, P. Tonella, W. Prasetya, P. M. Kruse, A. Bagnato, M. Harman, and O. Shehory, "FITTEST: A new continuous and automated testing process for future internet applications," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014*, S. Demeyer, D. W. Binkley, and F. Ricca, Eds. IEEE Computer Society, 2014, pp. 407–410. [Online]. Available: https://doi.org/10.1109/CSMR-WCRE.2014.6747206

[13] M. Greiler, M. D. Storey, and A. Noda, "An actionable framework for understanding and improving developer experience," *IEEE Trans. Software Eng.*, vol. 49, no. 4, pp. 1411–1425, 2023. [Online]. Available: https://doi.org/10.1109/TSE.2022.3175660

[14] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "SapFix: Automated end-to-end repair at scale," in *41st International Conference on Software Engineering*, J. M. Atlee and T. Bultan, Eds. Montreal: ACM, 25-31 May 2019, pp. 269–278. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2019.00039

[15] N. Alshahwan, "Industrial experience of genetic improvement in Facebook," in *GI-2019, ICSE workshops proceedings*, J. Petke, S. H. Tan, W. B. Langdon, and W. Weimer, Eds. Montreal: IEEE, 28 May 2019, p. 1, Invited Keynote. [Online]. Available: https://doi.org/10.1109/GI.2019.00010

[16] N. Alshahwan, J. Chheda, A. Finogenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Automated unit test improvement using large language models at Meta," in *FSE*. Porto de Galinhas, Brazil: ACM, July 15-19 2024, pp. 185—196. [Online]. Available: https://doi.org/10.1145/3663529.3663839

[17] K. Serebryany, "OSS-Fuzz - Google's continuous fuzzing service for open source software," in *26th USENIX Security Symposium*. Vancouver: USENIX Association, August 16-18 2017, retrived 8 October 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany

[18] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *USENIX Workshop at WOOT 20*. online: USENIX Association, 11 August 2020, p. 12, https://www.usenix.org/conference/woot20/presentation/fioraldi.

[19] gem5 Developers' Meeting: August 2024, "Discussion on bugs in gem5 from fuzzing with SearchSYS," https://github.com/orgs/gem5/discussions/1398, 8 Aug 4pm 2024. [Online]. Available: https://www.youtube.com/watch?v=hEyhXJg-rbU

[20] B. R. Bruce, A. Dakhama, K. Even-Mendoza, W. Langdon, H. Menendez, and J. Petke, "Artifact of Search+LLM-based Testing for ARM Simulators," Oct. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.13909721

[21] Zalewski Michal, "Technical "whitepaper" for afl-fuzz," http://lcamtuf.coredump.cx/afl/technical_details.txt, Retrieved April 21, 2023.

[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *NEURIPS*, 2017, pp. 5998–6008. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[23] L. Yang, C. Yang, S. Gao, W. Wang, B. Wang, Q. Zhu, X. Chu, J. Zhou, G. Liang, and J. Chen, "On the evaluation of large language models in unit test generation," in *ASE*, 2024, pp. 1607–1619. [Online]. Available: https://doi.org/10.1145/3691620.3695529

[24] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," in *NEURIPS*, vol. 35, 2022, pp. 22 199–22 213. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/8bb0d291acd4acf06ef112099c16f326-Paper-Conference.pdf

[25] ISO C, Working Group SC22/WG14, "The c17 standard for the c programming language (draft iso/iec9899:2017 of iso/iec 9899:2018), Index Section, pp. 476-515," https://www.iso.org/standard/74528.html, 2018.

[26] K. Even-Mendoza, A. Sharma, A. F. Donaldson, and C. Cadar, "GrayC: Greybox fuzzing of compilers and analysers for C," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. Seattle, WA, USA: ACM, July 17-21 2023, pp. 1219–1231. [Online]. Available: https://doi.org/10.1145/3597926.3598130

[27] D. Duplyakin *et al.*, "The design and operation of CloudLab," in *2019 USENIX annual technical conference (USENIX ATC 19)*, Renton, WA, USA, 2019, pp. 1–14. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/duplyakin

[28] SearchGEM5, https://github.com/karineek/SearchGEM5/tree/ssbse2023challenge/, Sept 2023.

[29] A. Herrera, H. Gunadi, S. Magrath, M. Norrish, M. Payer, and A. L. Hosking, "Seed selection for successful fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. Virtual, Denmark: ACM, 2021, pp. 230–243. [Online]. Available: https://doi.org/10.1145/3460319.3464795

[30] "AFL Internals - Stats, Counters and the UI," https://www.core.gen.tr/posts/007-afl-stats-counters-and-ui/, Fri, May 27, 2022.

[31] gem5, "Panic page table fault error in ARM when accessing invalid address in the simulated program," https://github.com/gem5/gem5/issues/1527, accessed: 2024-09-17.

[32] ——, "SIGABRT does not trigger its signal handler and skips outputs on ARM (won't fix)," https://github.com/gem5/gem5/issues/1544, accessed: 2024-09-17.

[33] ——, "Missing locale support within gem5 simulator on arm," https://github.com/gem5/gem5/issues/1547, accessed: 2024-09-17.

[34] ——, "Sim of localtime of time_t in ARM is different than native run," https://github.com/gem5/gem5/issues/1629, 2024, accessed: 2024-10-05.

[35] Personal Communication, "Bug in GMP 6.1.0 with ARMv8 (build and diff-tested with GCC-11 and CLANG-14)," October 2024, leading to some tests failed with both clang-14 and gcc-11.

[36] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2123–2138. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[37] S. Saha, L. Sarker, M. Shafiuzzaman, C. Shou, A. Li, G. Sankaran, and T. Bultan, "Rare path guided fuzzing," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, R. Just and G. Fraser, Eds. ACM, 2023, pp. 1295–1306. [Online]. Available: https://doi.org/10.1145/3597926.3598136

[38] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4All: Universal fuzzing with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24, Lisbon, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639121

[39] X. Jiang, Y. Dong, Y. Tao, H. Liu, Z. Jin, and G. Li, "ROCODE: Integrating backtracking mechanism and program analysis in large language models for code generation," in *Proceedings of the 47th International Conference on Software Engineering (ICSE)*. IEEE, 2025, accepted for publication. [Online]. Available: https://conf.researchr.org/track/icse-2025/icse-2025-research-track#Accepted-papers-First-and-Second-Cycle

[40] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015. [Online]. Available: http://dx.doi.org/10.1109/TSE.2014.2372785

[41] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, 2024, accepted on 27 August 2024. [Online]. Available: https://doi.org/10.1145/3695988

[42] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language model: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, vol. 50, no. 4, pp. 911–936, 2024. [Online]. Available: http://dx.doi.org/10.1109/TSE.2024.3368208

[43] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," in *ICSE-FoSE*, Melbourne, Australia, 14-20 May 2023, pp. 31–53. [Online]. Available: http://dx.doi.org/10.1109/ICSE-FoSE59343.2023.00008

[44] Q. Zhang, C. Fang, Y. Xie, Y. Zhang, Y. Yang, W. Sun, S. Yu, and Z. Chen, "A survey on large language models for software engineering," arXiv 2312.15223, 8 Sep 2024.

[45] R. Feldt, S. Kang, J. Yoon, and S. Yoo, "Towards autonomous testing agents via conversational large language models," in *ASE*, Luxembourg, 11-15 September 2023, pp. 1688–1693. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00148

[46] S. Yoo, "Executing one's way out of the Chinese Room," in *13th International Workshop on Genetic Improvement @ICSE 2024*, G. An, A. Blot, V. Nowack, O. Krauss, and J. Petke, Eds. Lisbon: ACM, 16 April 2024, p. viii, Invited Keynote. [Online]. Available: http://dx.doi.org/10.1145/3643692

[47] T. Le, T. Tran, D. Cao, V. Le, T. N. Nguyen, and V. Nguyen, "KAT: Dependency-aware automated API testing with large language models," in *ICST*, Toronto, 27-31 May 2024, pp. 82–92. [Online]. Available: http://dx.doi.org/10.1109/ICST60714.2024.00017

[48] J. Hu, Q. Zhang, and H. Yin, "Augmenting greybox fuzzing with generative AI," arXiv 2306.06782, 2023.

[49] A. Groce, R. van Tonder, G. T. Kalburgi, and C. Le Goues, "Making no-fuss compiler fuzzing effective," in *CC '22: 31st ACM SIGPLAN International Conference on Compiler Construction*. Seoul, South Korea: ACM, April 2-3 2022, pp. 194–204. [Online]. Available: https://doi.org/10.1145/3497776.3517765

[50] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "NAUTILUS: Fishing for deep bugs with grammars," in *Network and Distributed Systems Security (NDSS) Symposium*, San Diego, CA, USA, 24-27 February 2019. [Online]. Available: http://dx.doi.org/10.14722/ndss.2019.23412

[51] V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNET: A greybox fuzzer for network protocols," in *ICST*, Porto, Portugal, 24-28 October 2020, pp. 460–465. [Online]. Available: http://dx.doi.org/10.1109/ICST46399.2020.00062

[52] F. A. Endo, D. Couroussé, and H.-P. Charles, "Micro-architectural simulation of in-order and out-of-order ARM microprocessors with gem5," in *2014 international conference on embedded computer systems: Architectures, modeling, and simulation (SAMOS XIV)*. IEEE, 2014, pp. 266–273. [Online]. Available: http://dx.doi.org/10.1109/SAMOS.2014.6893220

[53] I. Wang, P. Chakraborty, Z. Y. Xue, and Y. F. Lin, "Evaluation of gem5 for performance modeling of ARM Cortex-R based embedded SoCs," *Microprocessors and Microsystems*, vol. 93, p. 104599, 2022. [Online]. Available: https://doi.org/10.1016/j.micpro.2022.104599

[54] Y. Qiu, S. Yi, M. Jing, X. Xiong, D. Xu, X. Zhu, X. Zeng, and Y. Fan, "Performance error evaluation of gem5 simulator for ARM server," in *2023 IEEE 15th International Conference on ASIC (ASICON)*. IEEE, 2023. [Online]. Available: http://dx.doi.org/10.1109/ASICON58565.2023.10396046

[55] N. Bruns, V. Herdt, D. Große, and R. Drechsler, "Toward RISC-V CSR compliance testing," *IEEE Embedded Systems Letters*, vol. 13, no. 4, pp. 202–205, 2021. [Online]. Available: https://doi.org/10.1109/LES.2021.3077368

[56] L. Bossuet, V. Grosso, and C. A. Lara-Nino, "Emulating side channel attacks on gem5: lessons learned," in *2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, Delft, Netherlands, 3-7 July 2023, pp. 287–295. [Online]. Available: http://dx.doi.org/10.1109/EuroSPW59978.2023.00036

[57] K. Serebryany, M. Lifantsev, K. Shtoyk, D. Kwan, and P. Hochschild, "SiliFuzz: Fuzzing CPUs by proxy," arXiv 2110.11519, 2021.

[58] R. Rajeev and X. Song, "An empirical study of fuzz stimuli generation for asynchronous fifo and memory coherency verification," *Journal of Electrical Electronics Engineering*, vol. 2, no. 3, pp. 302–306, 2023.