Automated Software Engineering (2025) 32:63
https://doi.org/10.1007/s10515-025-00531-7

Enhancing search-based testing with LLMs for finding bugs in system simulators

Aidan Dakhama[1] · Karine Even-Mendoza[1] · W. B Langdon[2] · Héctor D. Menéndez[1] · Justyna Petke[2]

Received: 30 August 2024 / Accepted: 29 May 2025 / Published online: 10 July 2025

Abstract

Despite the wide availability of automated testing techniques such as fuzzing, little attention has been devoted to testing computer architecture simulators. We propose a fully automated approach for this task. Our approach uses large language models (LLM) to generate input programs, including information about their parameters and types, as test cases for the simulators. The LLM's output becomes the initial seed for an existing fuzzer, `AFL++`, which has been enhanced with three mutation operators, targeting both the input binary program and its parameters. We implement our approach in a tool called `SearchSYS` . We use it to test the `gem5` system simulator. `SearchSYS` discovered 21 new bugs in `gem5` , 14 where `gem5` 's software prediction differs from the real behaviour on actual hardware, and 7 where it crashed. New defects were uncovered with each of the 6 LLMs used.

Keywords Fuzzing · Differential back-to-back testing · Systems · ISA x86 · LLM · Code LLM · Tiny LLM · ANN · SBSE · SBFT · Genetic improvement of tests · Gem5 · LLM in software engineering · TinyLlama · Phi2 · Llama2 · Magicoder · CodeBooga · GPT-3.5-turbo

1 Introduction

Testing plays a key role in software's lifecycle. Today, test generation is often expensive, tedious and labour-intensive. The task becomes even more difficult for emulators, that simulate different hardware. Creating a test suite for system-level simulation software is challenging. For example, `gem5` (Binkert et al. 2011) simulates software

Aidan Dakhama, Karine Even-Mendoza, W. B Langdon, Héctor D. Menéndez, and Justyna Petke contributed equally to this work.

Extended author information available on the last page of the article

execution on different architectures, either processor micro-architectures or system-level. Considering that the set of test inputs for such a simulator is a combination of both the architecture simulation *and* the program that runs within that simulation, the space of possible inputs for testing is exponentially large.

Previously (Dakhama et al. 2023), we proposed a novel way of testing system simulator software. Our differential testing (McKeeman 1998) approach uses large language models (LLMs) to first generate a set of initial programs, which are then compiled and fed through a fuzzer (a modified version of `AFL++`, see Section 3.2). This allows us to generate a large set of software that runs on specific hardware, as well as on a simulator that emulates the same computer architecture. If the outputs differ, we have potentially found a bug in the given system simulator (see Fig. 2). Each mismatch is flagged for further investigation.

Here we extend our approach (Dakhama et al. 2023), to make it fully automated and provide a more diverse set of test inputs, by: (1) generating new inputs directly via LLMs, including extracting software arguments and their types; (2) using a new mutator to modify software arguments' data types (e.g., mutating `0:INT32` to `55:INT64`); (3) improving fuzzing throughput; and (4) conducting an empirical study to evaluate our approach using 6 LLMs, with overall 70 different experimental combinations: 14 corpora and 5 configurations of `SearchSYS`. Our extended approach is presented in Fig. 1.

We implemented our approach in a tool called `SearchSYS` and used it to test the `gem5` (Binkert et al. 2011) software simulator. Overall, `SearchSYS` revealed 14 bugs that caused `gem5` to produce behaviour different from the one observed when the binary was run outside of `gem5`, as well as 7 crashes. Bugs were discovered by `SearchSYS`, regardless of the LLM used. They have all been reported to the developers.

To summarise, our contributions are:

1. A fully automated novel approach for testing software system simulators by combining large language models with fuzzing.
2. A prototype implementation of our approach, named `SearchSYS`.

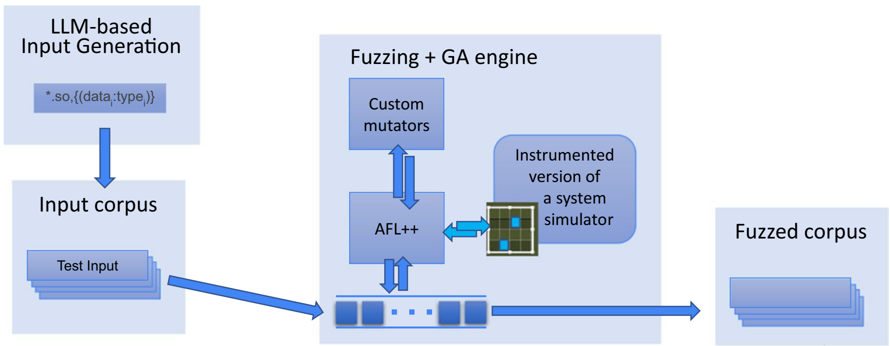

Fig. 1 High-level description of `SearchSYS` for automatically testing simulation software

3. An extensive empirical study using `SearchSYS` to test the `gem5` system simulator. The LLM-generated inputs led to the discovery of 369 issues — this number increased to 101 442 when the LLM-generated programs' binaries and their arguments were seeded to `SearchSYS` 's fuzzer. These findings reveal 14 new separate behaviours that differ from running on native hardware and 7 new independent crashes.
4. An investigation into the effect of 6 large language models and 5 `SearchSYS` configurations on the effectiveness of `SearchSYS`.

Section 2 presents the background required to understand our contribution; Section 3 presents our approach for testing computer system simulator software. Section 4 presents the research questions we pose to evaluate our approach. Section 5 presents our methodology to answer our research questions, while Section 6 presents our empirical results, and Section 7 discusses them. We present a threats to validity section addressing potential limitations of our findings in Section 8. Additionally, we provide related work in Section 9 and overall conclusions in Section 10.

To facilitate the replicability of our study, we make our artifact freely available (Dakhama et al. 2024).

2 Background

Our approach to testing system simulators combines technologies from two disciplines: search-based software testing, in the form of fuzzing, and generative AI in the form of large language models. In this section, we describe these two technologies as well as provide background on the application of our approach, i.e., detection of errors in computer system simulators.

2.1 System simulators and silent errors

Simulators are broadly categorised by the domain or type of systems to be simulated. Here, we are interested in computer-system architecture simulators, such as `gem5` (Binkert et al. 2011). These emulate software execution in specific architectures, e.g. for testing and verifying software behaviour within these computers or operating systems even before they physically exist.

Although `gem5` is the system-under-test (SUT) in the evaluation, discussed further in Section 5, our findings apply to computer-system architecture simulators in general. For brevity throughout the rest of the paper, we use 'simulator' for the computer-system architecture simulator and 'binary' for the input software it emulates.

In simulators, bugs manifest in various forms, including system hangs, crashes, and discrepancies between simulation and native runs. Bugs encompass cases where the native run terminates correctly, but the simulation crashes, and cases where the simulator fails to replicate a crash in the native environment. Silent errors, or *mis-simulations*, further complicate matters as they occur when a simulation runs without overt errors but produces incorrect results silently. Silent errors are challenging to detect because the correct result is commonly unknown, cf. the Oracle Problem (Barr et al. 2015). To address this concern, we employ differential testing, using the native system as our reference point to detect missimulations (McKeeman 1998; Chen et al. 2020).

2.2 Fuzzers

Fuzzing is a software testing technique that automatically generates test inputs, usually aiming to reveal crashes and/or increase code coverage of the *software-under-test* (SUT).

We use the `AFL++` fork (Fioraldi et al. 2020) of the American Fuzzy Lop (`AFL`) fuzzer (Michal 2023). `AFL++` instruments the SUT by compiling it with additional instructions for obtaining feedback about code coverage. When fuzzing, `AFL++` takes an initial corpus of tests (known as seeds) and the instrumented SUT. The fuzzer uses test coverage information in the instrumented version[1] of the SUT to explore new inputs by executing the target program with seeds and evaluating the coverage achieved. Subsequently, the fuzzer applies diverse mutations to a queue of input seeds to discover new program execution paths and thereby increase code coverage in the SUT. Inputs discovering new branch transitions are added to the fuzzer's queue until reaching the termination condition, such as achieving the desired coverage or reaching a time limit. AFL is configurable via its Python and C/C++ APIs.[2]

Seeds are typically input data for the SUT. However, in the context of this work, seeds represent a combination of executable binary program files and their arguments, rather than plain input data, and the SUT is a simulator. We discuss in detail our choice of seed representation for fuzzing simulators in Section 3.

2.3 Code coverage

In testing, code coverage measures are used to approximate the portion of a program's code that is executed with given test inputs, such as line, function, or branch coverage. In this work, during fuzzing and corpus minimisation, `AFL++` and `afl-cmin` apply"binned hitcounts", a simplified version to estimate branch (edge) coverage. We refer the interested reader to the original work and recent analysis of `AFL++` coverage (Michal 2023; Fioraldi et al. 2023). Outside the fuzzing campaign, we use line coverage, which measures the executed lines of code during SUT execution to address the code coverage aspects of RQ1 and RQ4. Line coverage is expressed as either an absolute value (e.g. 20 lines executed) or as a percentage relative to a baseline or previous results (e.g. 5% increase over a baseline of 20 lines). We use the former option in our evaluation, similar to Even-Mendoza et al. (2023); Xia et al. (2024).

[1] Instrumented gem5 code is automatically generated during compilation by using the GNU Compiler Collection (GCC) gcov profiling tool https://gcc.gnu.org/onlinedocs/gcc/gcov/introduction-to-gcov.html.

[2] https://aflplus.plus/docs/custom_mutators/, and environment variableshttps://aflplus.plus/docs/env_variables/, which allow for customisation and tuning of its behaviour. We leverage these options to build SearchSYS.

2.4 Artificial intelligence: Large Language Models (LLMs)

Large language models (LLMs) are increasingly prevalent in artificial intelligence (AI) research, e.g., to aid text generation, including the generation of source code. The emergence of GPT models, based on transformers with attention mechanisms, has been widely accepted, leading to the development of new models aiming to surpass their capabilities. Among the most prominent GPT versions are `GPT-3.5` (Dale 2021), publicly available through `OpenAI` 's framework, and `GPT-4` (Achiam et al. 2023), accessible via private subscription. Recent large language models competing with `OpenAI` 's offerings at an industrial level include `Llama2` (Touvron et al. 2023), developed by Meta and currently available to the public, and `Bard` (now renamed to `Gemini`) (Aydin 2023), owned by Google and accessible through its framework. Additionally, various research communities have produced new open-source text generation models, such as `Dolphin` and `Mistral`, which are available on the `HuggingFace` platform (Jain 2022).

Several LLMs have been recently published for the specific task of program source code generation (aka Code LLMs Pahune and Chandrasekharan 2023). These are based on general-purpose architectures or similar training techniques. We evaluate several different LLMs. In addition to `ChatGPT-3.5`, we use `Phi2` (Javaheripi and Bubeck 2023) (from Microsoft, designed for software generation), `CodeLlama` (Grattafiori et al. 2023) (which is derived from Llama specifically for source code generation), `Magicoder` (Wei et al. 2023) (which combines transformers and auto-encoders for code generation) and `CodeBooga`. Notably, `CodeBooga` (Code Booga n.d.) demonstrates how novel techniques, like BlockMerge Gradient, can amalgamate knowledge from different LLMs. Specifically, `CodeBooga` combines `Phind-CodeLlama-34B-v2` and `WizardCoder-Python-34B-V1.0` (`Phind-CodeLlama-34B-v2` outperformed `GPT-4` in benchmark evaluations Gu et al. 2024). They operate within the `Ollama` framework, which helps to set them up and ensure they run under standard conditions.

3 `SearchSYS`: Testing system simulators

Our approach for testing simulators follows the next steps:

1. **Test Input Generation** First, we use an LLM to generate a set of programs. In our previous work, we began by using existing programs. In this paper, we remove the need for such examples (Section 3.1.2).
2. **Coverage-Guided Mutation-Based Fuzz Testing** A single test input for the simulator is composed of an executable program binary file (--binary) and its arguments (--options). We thus compile the LLM-generated programs and input them into a fuzzer to generate diverse variants of our test inputs (Section 3.2).
3. **Differential Testing** Finally, we compare the result obtained from running our test inputs through a simulator with the outcome of the actual test execution in a native environment (Fig. 2).

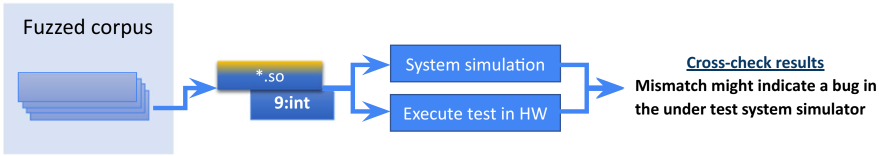

Fig. 2 Differential testing of system simulators involves the use of a mutated corpus of test programs to identify discrepancies between the output generated by a software simulator and a reference standard: direct execution

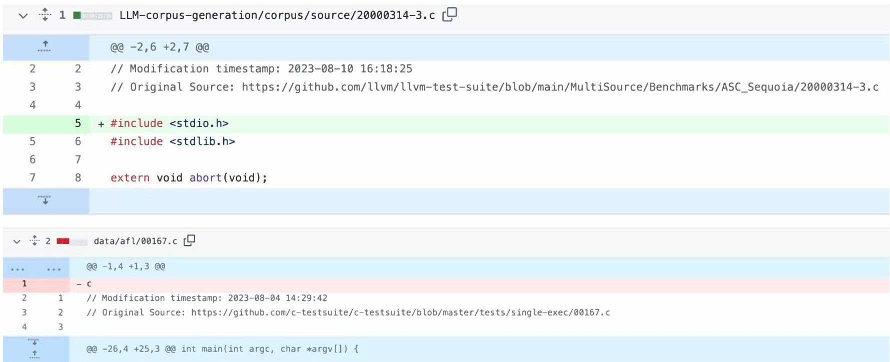

Fig. 3 A sample of the changes using our `Bash` scripts to fix minor errors of LLM-generated programs as shown in `SearchGEM5` (Dakhama et al. 2023)'s GitHub

The following subsections detail each of the aforementioned steps of our approach.

3.1 LLM-based test inputs generation

We create a corpus of *parameterised test inputs*. To execute a single test in the simulator under test, we need: the program binary to simulate; its arguments; and their types (e.g. 32-bit int). We use LLMs to generate all components necessary for creating a test input. The process includes the binary creation by compiling the test program source code obtained from the LLM. Additionally, we prompt the LLM to provide a file containing the test program source's arguments and their types.

3.1.1 Generation via test program sources

In our preliminary work (Dakhama et al. 2023) we generated parameterised test inputs from test suites and tutorial C programs via LLMs.

We used a `Bash` script to amend minor errors, such as adding missing includes and removing empty lines or free text at the start or end of C programs. For instance, in Fig. 3, we automatically fixed two programs; we inserted a missing `stdio.h` include to the test input `20000314-3.c`, while we removed a free text not in a comment from the test input `00167.c`.

We applied a few-shot prompting technique (Brown et al. 2020). We had three prompts to facilitate the LLM's comprehension of the task: 1) a simple prompt that describes the task using a small C code and a free text description; 2) a prompt that gives an example of a good response; and 3) a prompt that gives an example of a wrong response with a short explanation of what is not valid.

To enable this prompt to handle many programs simultaneously, we extended it by appending the prompt with many programs as illustrated in Listing 1. While this approach helps guide the LLM's comprehension, it may also limit the diversity of the responses generated, while also limiting the throughput of the generation due to the large number of tokens required in the prompt.

```
"I will give you a set of N programs from source X, can you generate
 a pair per program with an input sample and its type information for the
 second program? These are the programs: (name: code, name: code,...)".
```

Listing 1 Prompt used for generating input samples and type information in Dakhama et al. (2023).

The `GPT-3.5-turbo` returned pairs of programs, consisting of the original C program and its parameterised counterpart, plus, for each argument, an example of a valid argument (input value) and its type (e.g. `5 INT32`). (The original program is for sanity checks and types are needed by the tool for input mutation.) The programs produced by the LLM that have no arguments or fail to compile were deemed invalid.

With this approach, we gained better control over test input generation by constraining the LLM to a predefined set of C programs. However, it yielded a low rate of valid programs for fuzzing with `AFL++`. In Section 3 of Dakhama et al. (2023) (our previous work), we observed that `GPT-3.5-turbo` heavily relied on external program sources and required additional adjustments such as our `Bash` script to prepare the final compiled program.

Zero-Shot Over Few-Shot: The change to zero-shot prompting in Section 3.1.2 improves efficiency, scalability, and diversity of the test input generation by removing the dependency on predefined examples and Bash script's fixes and addressing limitations of a few-shots' time-consuming, semi-manual, and error-prone process. This enables fully automated fuzzing, eliminating human involvement in seed generation and selection, which is crucial for large-scale, continuous testing without human bottlenecks to achieve high throughput and test input diversity.

3.1.2 Zero-shot prompting test input generation

We apply zero-shot prompting for test input generation instead of a few-shots approach as in our previous work (Dakhama et al. 2023) to address efficiency and scalability challenges while enabling broader and more diverse test input generation.

We augment the prompts with four C language-related token categories instead of relying on test suites and C programs. We use three types of tokens: (1) program names from tutorials (e.g., "Hello World"), (2) compiler optimisations (e.g., "Dead Code Elimination") or components (e.g., "Handles Abstract Syntax Tree (AST)"), and (3) phrases from the ISO's C standard (ISO C, Working Group SC22/WG14 2018) (e.g., "asinh function"). Note that we scan the last section of ISO C, Working Group SC22/WG14 (2018) into our generator to be able to generate these phrases.

A phrase can be `initialised` (first word, ISO C, Working Group SC22/WG14 2018, page 490).

`SearchSYS` starts with a single prompt setting the LLM role, outlining the task's context to generate the desired output using a similar prompt as in Section 3.1.1. However, we set the role once, rather than repeating this process every several prompts. Additionally, we do not test if the model adheres to a specific input program nor give examples of good and bad responses. Following this stage, the LLM is presented with an automatically generated prompt with a subset of three tokens selected randomly. Table 1 describes the three token types (Type Col.) across four C language-related token categories (Category Col.) with a few examples (Example Col.).

Subsequently, we create random queries following a straightforward pattern, as shown in Listing 2.

```
"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 2 A *template prompt* for generating C test program source using random tokens from Table 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 Vectorization
optimisations, covers this part of the compiler Frontend, and exercises
this idea in C: signed type. To recap the code contains these
Vectorization, Frontend and signed type;"
```

Listing 3 A *concrete prompt* for generating C test program source using the tokens "Vectorization," "Frontend," and "signed type".

This process achieves fully automated test input generation. Figure 4 shows a generated parameterised C program created using CodeBooga which includes `Vectorization`, `Frontend` and `signed type` tokens, crafting the concrete prompt in Listing 3. The test input, as before, includes: parameterised C programs, an example of a valid argument (input value) and its type for each argument (e.g. `9:INT32`). For full details see our tool artifact.

Table 1 C language-related token categories for filling template gaps in listing 2

ID	Type	Category	#Tokens	Examples
1	Token-1	Compiler Optimizations	26	"Scalar Optimizations","Dead Code Elimination","Constant Folding"
2	Token-2	Compiler Parts	36	"Frontend","Sema","Serialization","Parse","Lex","AST"
3	Token-3	Domain Problems	192	"C Program to Sort an Array using Merge Sort","Calendar Year in Different Formats","input includes several arguments"
4	Token-3	Compiler Standard Indices (ISO C, Working Group SC22/WG14 2018)	3480	"AND operators","cimagl function","EOF","locale","pow","SCNiMAX"

````
```c
#include <stdio.h>
#include <stdlib.h>
#include <immintrin.h>        // include header for AVX intrinsics
int main(int argc, char *argv[]) {
  if (argc != 2) {
    printf("Usage: %s <integer>\n", argv[0]);
    return -1;
  }
  int input = atoi(argv[1]);
  // load input into vector register
  __m256i vinput = _mm256_set1_epi32(input);
  // square the integer using AVX instruction
  __m256i vresult = _mm256_mullo_epi32(vinput, input);
  int result[8];
  // store vector register to memory
  _mm256_storeu_si256((__m256i*)result, result);
  printf("Square of %d is: %d\n", input, result[0]);
  return 0;
}
```
````

Fig. 4 C code generated by LLM CodeBooga and the concrete prompt in Listing 3. Note: the frontend program takes inputs from user and send answer back to the user, the use of vector instructions, e.g. `_mm256_set1_epi32`, and signed types, e.g. `int input`

This streamlined approach generates inputs at a significantly higher rate, compared to previous work, in a way that does not depend on which LLM is used. It also provides high-level control over test case generation of parameterised test inputs. In Section 6.1, we evaluate the throughput of different LLMs, including `GPT-3.5-turbo`. For brevity, we refer to Section 3.1.1's approach as the *Semi-manual Approach* and Section 3.1.2's as the *Template Prompt Approach* throughout the rest of the paper.

3.2 Fuzzing

`AFL++` operates by taking an initial set of files, each serving as an input to the software-under-test (SUT) and instrumenting the SUT to measure test coverage. This section describes our extension of `AFL++`, `SearchSYS`, for testing simulators with complex test inputs, using a coverage-guided mutational approach. We further discuss applications of `SearchSYS` to test a specific simulator, `gem5`, in Section 5.

We have extended `AFL++` by writing our mutation operators and part of the mutation strategy, although we still make use of `AFL++` 's coverage selection criteria. This enables the selection of specific test inputs from the corpus and mutation of their binary file, arguments, or types, based on the test coverage data collected by `AFL++` for each test input. We do not use `AFL++` 's built-in mutation operations at all because they are not suitable for fuzzing binary files and types. These operations often disrupt the necessary structure required to create valid (or semi-valid) test inputs, resulting in mostly broken and useless test inputs. Instead, we use `AFL++` 's custom mutators feature to load and apply our own mutators.

In our previous work (Dakhama et al. 2023), we introduced `SearchGEM5`, an automated testing technique that combines LLMs with search-based testing for system simulator testing. We used `GPT-3.5-turbo` for parameterised C program generation, and we discussed our extension to any LLM model in Section 3.1. Here, we extend the search-based testing aspect of Dakhama et al. (2023), which only included diversifying the arguments' values, the generation of new test inputs via direct fuzzing of their executable program binary file, and the selection of test inputs to be fuzzed based on coverage feedback.

To further enhance our approach, we have made some additional improvements:

- **New Mutator Operator.** We introduce a new mutator that varies the type of arguments.
- **Independent Mutators.** We implement and load to `AFL++` each mutator separately, allowing coverage feedback to influence the selection of test inputs, while also allowing the use of different mutator operators from the three available ones, thereby enhancing the search-based testing capability of `SearchSYS`.
- **Improved Fuzzing Throughput.** We improve fuzzing throughput by no longer relying solely on `AFL++` 's filtering of test inputs. Instead, we save all test inputs for later inspection, including additional scripts for differential testing, similar to the approach in Even-Mendoza et al. (2023).

Next, we describe in detail each of these contributions.

3.2.1 Custom `AFL++` mutation operators

While bit-flip mutation is a standard feature in `AFL++`, its application must consider the context. Introducing mutations that result in binaries being unable to load or execute a single instruction leads to inefficient testing of the SUT, making it unlikely that developers will be prepared to spend effort on bug identification or bug fixing.

Consequently, we implement a new bit-flip mutation operator that controls the number of bit-flips and frequency of application while limiting it to a program's compiled binary file. We do not apply bit-flip to arguments or type information to preserve the structure of the test input. In Section 7.1, we analyse a fuzzed test input where a bit-flip modifies a pointer value (address) without impacting instruction validity, an unlikely outcome using the standard `AFL++` bit-flip operator. This modification caused a mismatch between `gem5` and the simulated architecture.

We have introduced three mutation operators of a test input for testing system simulators: 1) bit-flip operator to edit a program's compiled binary file, 2) a range-enhanced operator to edit argument values within their specified type range and 3) an operator to edit the value's type. Operator (2) uses type information to ensure that the arguments remain valid. In contrast, operator (3) mutates the type itself, such as changing from `INT32` to `LONG`, at random, which can potentially expose memory safety issues in the SUT. We currently support all integer types, floating-point numbers, doubles, and strings. However, we have not yet implemented support for pointers. Figures 5 and 6 show examples of arguments' value (operator (2)) and type mutations (operator (3)), respectively.

We load all three mutators (1-3) using the `AFL++` option, allowing `AFL++` 's heuristics to select the next mutation operator. However, we intervene by decreasing the probability of choosing (2), as `AFL++` favours this operator due to its low risk of failing, which is too conservative for a fuzzing approach.

```
./mutator_args.so test.o, 1:INT, 2:LONG, "Hello":STRING
After Mutation:   test.o, 10:INT, 2:LONG, "universe":STRING
```

Fig. 5 Example of mutation operator (2) changing two argument values: first argument changes from `1` to `10`, and third argument from"Hello"to"universe"

```
./mutator_types.so test.o, 1:INT, 2:LONG, "Hello":STRING
After Mutation:    test.o, 1:LONG, "2":STRING, "Hello":STRING
```

Fig. 6 Example of mutation operator (3), which changes the first argument's type from `INT` to `LONG` and the second argument's type from `LONG` to `STRING`

`SearchSYS` implements the extension to `AFL++`. It evaluates new test inputs in the form of `binary name, arguments list, types`. `SearchSYS` then uses this information to carry out our mutation operators either directly to the compiled binaries or to their arguments.

3.2.2 Fuzzing throughput improvements

`AFL++` discards test inputs (1) that exhibit no crash or hang and (2) do not contribute to coverage. However, such inputs can still uncover missimulation in the system simulator under test, thus removing them from the queue degrades `SearchSYS` 's ability to detect missimulations.

We incorporate the following idea in `SearchSYS`: fuzzing and differential testing are carried out separately, similar to the approach by Even-Mendoza et al. (2023). Consequently, we save these fuzzed test inputs for later examination. After fuzzing, we employ differential testing, between running directly on x86 hardware (i.e., native) and being run by the simulator. In Section 6, we analyse the efficiency of this idea in terms of bug finding and coverage.

Another factor reducing fuzzing throughput is related to how many mutation operations `AFL++` performs in a single fuzzing iteration. `AFL++` 's `afl_custom_fuzz_count` parameter controls the number of times a test input should be mutated and executed against the target. The iteration fails if any attempt to mutate the test input is unsuccessful, and the resultant fuzzed test input is then discarded. An attempt using Operator (1) (bit-flip) is more likely to fail than Operator (2) or Operator (3) because it involves modifying a compiled binary file.

We customise the `afl_custom_fuzz_count` to control the number of mutation attempts per iteration. A lower value reduces the likelihood of an iteration failing. A too-low value leads to inefficient fuzzed input generation due to the overhead each iteration introduces.

Note that `afl_custom_fuzz_count` is a hyperparameter of `AFL++`, not `SearchSYS`. In Section 6, we investigate the optimal value of `afl_custom_fuzz_count` in the context of system simulator fuzzing, measuring the improvement in throughput, coverage, and bug-finding.

4 Research questions

Next, we state our research questions, which guide our `SearchSYS` evaluation.

To assess the reliability of LLMs as a source of test inputs for a system simulator, when given the prescribed test framework (as outlined in Section 3), our objective is to evaluate:

> **RQ1:** To what extent can different LLMs effectively generate parameterised C programs for testing system simulators that adhere to the specified requirements?

During the test generation process, LLMs may generate similar test cases or test cases that do not increase code coverage. Considering that this negatively impact the fuzzer's performance, we minimise the test suite size through a test selection process. This also allows us to evaluate:

> **RQ2:** What is the level of test case redundancy related to branch coverage in the test suite generated by the LLMs?

Does fuzzing (concretely with `AFL++`) enhance the basic test coverage achieved by LLM-generated test suites in a system simulator, after obtaining a minimised test suite? Which LLMs perform best in achieving high coverage with `AFL++` diversification? Considering that the search space of every possible binary program and input for a system is vast, we adapted `AFL++` to construct `SearchSYS` (Section 3).

`SearchSYS` has parameters that affect the fuzzing process. These can be encoded into different *configuration files* for `SearchSYS` (Section 6.3). We randomly generated 30 settings files. Consequently, we ask:

> **RQ3:** What are the most suitable parameters for running `SearchSYS`?

Are there any customisations which have a particularly positive effect? Since fuzzing is non-deterministic, how significant is the impact of the parameter configuration, particularly when combined with our custom mutators, on the use of our approach in practice? Furthermore, we aim to quantify the impact of our `AFL++` 's customisation, as outlined in Section 3.2. Beyond the technical aspects of `AFL++` 's ability to instrument and fuzz system simulators, we ask:

> **RQ4:** How significantly does our customisation enhance `SearchSYS` 's efficacy in bug finding and coverage improvement?

5 Methodology

Here, we present our methodology to answer our research questions.

5.1 gem5 Use case

Our case study, conducted during the evaluation phase of our research, focused on the instrumentation, testing, and examination of the *Instruction Set Architecture* (ISA) x86 part of gem5. gem5 is an open-source simulator widely used in both academia and industry. To facilitate search-based fuzz testing, we had to instrument gem5 and address potential performance overheads and scalability issues that arise when simulating large-scale computer systems.

5.1.1 Logic circuit simulator

gem5 is a state-of-the-art discrete time simulator for logic circuits. It is commonly used to test the logic design of new electronic components, such as memory cache systems, field-programmable gate arrays (FPGAs), and even CPUs. gem5 is a large open-source project hosted on GitHub, with a primary coding language being C++ and Python. We use the version provided in the 2023 SSBSE Challenge Track. It is based on gem5 staging branch v23.0, but incorporates the latest gem5 features and improvements into a stable release. The tool also includes objects, shared libraries and images, occupying over 28 GB of memory. It consists of $\sim$ 1.34 million lines of code, with more than a million lines written in C++. gem5 is controlled by a Python script, specified via its command line, which determines, for example, the size and type of memory the simulated binary can read and write to.

To facilitate comparison with our previous work (Dakhama et al. 2023), we employ the same modified version of an example file provided by the SSBSE Challenge Track 2023 organisers, i.e., hello-custom-binary.py, which we adapted to accommodate the specific requirements for SearchSYS 's test inputs, including file usage and additional parameters.

5.1.2 Instrumentation and AFL++ setup

During fuzzing with AFL++, to ensure consistent execution of test inputs, a Python script is used to communicate between AFL++ and gem5. We use SCons version v4.4.0 and the AFL++ fork (Fioraldi et al. 2020) of the American Fuzzy Lop (AFL) fuzzer (Michal 2023), version afl-fuzz++4.08c, GitHub commit f596a297.

SearchSYS employed AFL++ as the search engine, using its default parameters, except for two custom settings: the AFL++ 's map size was set to 1 200 000, with a time limit of 99 seconds and memory limit of 50 000 megabytes per test case execution. We customised the functions afl_custom_init, afl_custom_deinit, afl_custom_fuzz_count and afl_custom_fuzz to implement the customisations described in Section 3.2. To further diversify the test inputs, we set AFL_SHUFFLE_QUEUE to 1. The full AFL++ 's setup is described in SearchSYS 's artifact (Dakhama et al. 2024).

5.2 Selected LLMs

We selected `ollama` models based on their diversity, covering both general-purpose and code models, as well as a range of sizes determined by the number of Artificial Neural Networks (ANNs) parameters. The number of ANN parameters typically indicates the complexity and size of the neural network model. In LLMs, the model's size is correlated with the number of parameters, including weights and biases in the neural network architecture.

We explore how model size impacts fuzzing outputs by examining a new type of language model called small language models (SLM), also known as Tiny LLMs (Matricardi 2023). These models are smaller versions of LLMs, typically with parameter counts significantly smaller than standard LLMs. We employ two SLMs, `TinyLlama-1.1B` and `Phi2`, to assess the influence of model size on our fuzzing methodology in our evaluation. `TinyLlama-1.1B` is a general-purpose 1.1×10^9(1.1B) parameter language model, pre-trained on approximately 1 trillion tokens (Zhang et al. 2024). `Phi2` is a 2.7×10^9(2.7B) parameter language model designed for question answering, chat, and code generation (Javaheripi and Bubeck 2023).

We employed 6 language models listed in Table 2, along with a version of `GPT-3.5-turbo` previously used in our work (Dakhama et al. 2023). To address RQ2, we applied `AFL++` 's corpus minimisation (`afl-cmin`), which reduces the number of test cases it uses based on previously covered transitions. This allowed us to compare fuzzing performance and behaviour across different LLMs and methods. We generated a total of $2 \times 7 = 14$ input corpora, with two variants per LLM: one containing corpus minimisation (`afl-cmin`) and the other without. Subsequently, we compared the reduced and original test suites. To distinguish between these corpora, we have assigned them different names, for example, **Phi2** and **Phi2-cmin**.

5.3 Configurations

We consider 5 configurations in our evaluation when fuzzing:

Table 2 Language models for LLM-based input generation phase

Model	Type	Size	Version	Test Input Generation Methods
`TinyLlama`	SLM,L	637 MB	Git commit `2644915`	Zero-shot, Corpus Min.
`Phi2`	SLM,L	1.6 GB	Git commit `e2fd632`	Zero-shot, Corpus Min.
`Llama2`	L,G	3.8 GB	Git commit `78e2641`	Zero-shot, Corpus Min.
`Magicoder`	L,C	3.8 GB	Git commit `8007de0`	Zero-shot, Corpus Min.
`CodeBooga`	L,C	19.0 GB	Git commit `05b83c5`	Zero-shot, Corpus Min.
`GPT-3.5-turbo`	LLM,G	NA	ChatGPT (Feb. 11, 2024)	Zero-shot, Corpus Min.
`GPT-3.5-turbo` (SSBSE 2023)	LLM,G	NA	ChatGPT (Aug. 3, 2023)	Prog. Src., Corpus Min.

Type: C (code model), G (general-purpose model), L (local model), LLM (large language model), SLM (small language model). Methods applied to create the corpus: Prog. Src. (Input corpus generation via test program sources, Section 3.1.1), Zero-shot (zero-shot prompting, Section 3.1.2), Corpus Min. (corpus minimisation with `AFL++` -cmin)

1. `SearchSYS`. As introduced in this paper (with all improvements).
2. **Throughput Improvement by Keeping all Fuzzed Inputs.** We adapt `SearchSYS` to operate **solely** with the first throughput improvement option, as described in Section 3.2, namely fuzzing by proxy.
3. **Throughput Improvement with Optimising** `afl_custom_fuzz_count`. We adapt `SearchSYS` to operate **solely** with the second throughput improvement option, as described in Section 3.2, namely overriding `AFL++` custom function `afl_custom_fuzz_count`.
4. **No Throughput Improvement.** We adapt `SearchSYS` to operate without any throughput improvement options.
5. `SearchGEM5`**-SSBSE-2023.** `SearchGEM5` as in our previous work (Dakhama et al. 2023).

We use these configurations to assess the impact of each customisation.

Besides comparing with `SearchGEM5` (Configuration 5), we have constructed weaker versions of `SearchSYS`. Configurations 2-4 incorporate the type mutator and enable loading each mutator into `AFL++` separately. However, these configurations differ in their throughput improvement options. Configurations 2-3 exclude one option each: Configuration 2 employs fuzzing by proxy, saving all test inputs for later review, while Configuration 3 overrides the custom function `afl_custom_fuzz_count`. `Configuration 4 includes none`. In contrast, Configuration 1 includes all three customisations. This allowed us to isolate and evaluate the impact of each addition in `SearchSYS` relative to the baseline, `SearchGEM5`.

5.4 Experimental procedure

The complete flow of the experimental procedure is illustrated in Fig. 7.

For RQ1, we generated the input corpora running LLM-based input generation for 25 hours. We set the timeout per query (or prompt) to be 100 seconds. This decision dictated our selection of LLM models for test input generation, as models consistently reaching the timeout limit are unsuitable for the auto-generation task due to their expected low throughput rate. For RQ2, we applied corpus minimisation using `afl-cmin` of `AFL++`. This part is illustrated as the pre-fuzzing stage in Fig. 1.

For RQ3, we generated 30 sets of numbers, each containing three randomly selected values between 1 and 100 (i.e. randomly and uniformly over integers between 1 to 100), representing different parameter options for `SearchSYS` that affect the fuzzing process. This triplet format corresponds to the value of `afl_custom_fuzz_count` per each of the three mutators in Section 3.2. Each triplet is stored in a separate settings file, resulting in 30 files being used in our evaluation. RQ3 aims to select a single settings file for use in Configuration 1 and 3 during fuzzing (RQ4).

Finally, for RQ4, where we generate the final fuzzed corpus, we fuzzed each combination of input corpus (with and without minimisation) and configuration for 24 hours. This process was repeated 10 times to obtain more accurate results (Klees et al. 2018). We ran a total of 70 combinations: 14 different input corpora and 5 configurations. Our evaluation presents results at the granularity of each combination,

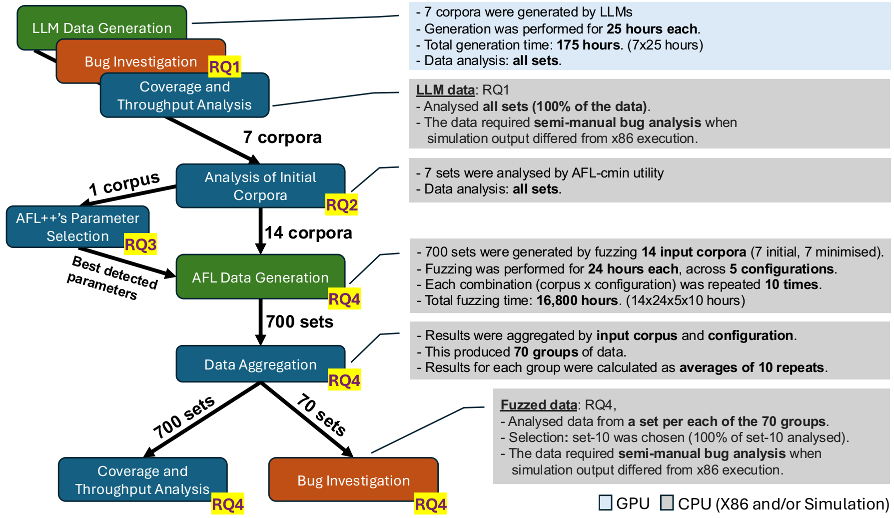

Fig. 7 Overview of the experimental procedure. (Note about RQ4 Bug Investigation - Set 10: `day_to_analyse = uniform_random_Int([1, 10])`, yielded `day_to_analyse = 10`. We used 100% of Day 10 data to construct Tables 7 and 8. The data is archived as `set-10.zip` in our Zenodo record)

Table 3 Computer hardware and software used during evaluation stages

Experiment	GPU?	Computer Specification
LLM-based Input Generation Local	Yes	Ubuntu Server 20.04.1 LTS, 256 GB RAM, NVIDIA A30 Tensor GPU of 24 GB, 80 logical CPUs
LLM-based Input Generation Remote as Service	No	Ubuntu 22.04.4 LTS, 32 GB RAM, i7-1185G7
Fuzzing & Coverage	Yes	Red Hat 8.5.0-20, 256 GB RAM, NVIDIA A30 Tensor GPU of 24 GB, Intel Xeon Silver 4316 CPU with 80 logical CPUs

averaging the 10 repetitions for each combination. Overall, a total of 700 of 24-hour runs were performed. This part is illustrated as the fuzzing + genetic algorithm (GA) engine part in Fig. 1. We note that AFL++ uses a GA internally.

To explore the bug detection capabilities of the generated test inputs, we applied differential testing, post-fuzzing. An overview of our post-fuzzing experiment procedure is illustrated in Fig. 2.

5.5 Experimental environment

We used different machines for our evaluation, some equipped with GPUs and others with only CPUs (x86_64), listed in Table 3.

LLM-based input generation The GPT-3.5-turbo-based generation used GPUs on servers running in data centres managed by OpenAI. For the rest of the language models, we implemented a small driver written in Java, querying Ollama models using the Ollama4j infrastructure. We used Ollama version 0.1.22 and Ollama4j version 1.0.44. The timeout per query was 100 seconds. We ran the LLM-based input generation method on the machines listed in Table 3, rows 1-2 for Ollama models and GPT-3.5-turbo, respectively.

Fuzzing and coverage We fuzzed gem5 using Docker with Debian-12 containers emulated with Podman (v4.6.1), using the machine specification in row 3 of Table 3, including running AFL++ for the corpus minimisation and parameter selection. To answer our research questions (e.g. RQ4 setting), we ran 70 docker containers in parallel for the fuzzing campaigns, each of which was a different combination of configuration and input corpus (5 x14) for 10 days by leveraging the configuration option AFL_NO_AFFINITY=1 from AFL++ to run the system in multiple cores. It attached a container per processor, keeping 10 processors for the host's other tasks. The experiments did not consume more than 130GB of RAM in total. The SUT was instrumented and compiled with afl-cc, GCC−11 and the default settings of gem5. The binary files of test inputs were compiled with GCC−11 and-O3. We measured the coverage of the input and fuzzed corpora with gcov. We built gem5 with g++ 11-O1 and gcov, adding gcov instrumentation overheads. We measured a smaller part of the gem5 codebase, i.e. that is relevant only to x86. We used the gcov-based tool gfauto (gfauto n.d.) to generate the coverage results in a human-readable format for 3370 files in the gem5 codebase (including header and system header files).

6 Results

We evaluated SearchSYS on its test generation capabilities, coverage, bug finding, and the efforts required to reproduce the results. We provided further details about the discovered bugs online at Dakhama et al. (2024).

6.1 RQ1: LLM test input generation effectiveness

To evaluate the suitability of LLMs as sources of inputs for fuzzing system simulators, we conducted 25 -hour runs with different LLM models. We assessed the effectiveness of our LLM-based Test Inputs Generation on 6 different language models (Section 5.2). We examined the test input throughput, coverage, and the capability to discover new bugs in the system under test (SUT). Data was collected from 6 LLMs, each generated through 25 hours of test input generation using LLMs. The initial corpus GPT-3.5-turbo (SSBSE 2023) is taken from Dakhama et al. (2023).

Programs extraction process Large models tend to generate proper code (i.e. complete programs) while smaller models often produce responses that are neither valid programs nor demonstrate a clear understanding of the question. Consequently, LLM test program generation does not follow the assumption of "one prompt equals one program". When LLMs generate an output, it can be either proper code or plain text. If the output contains code, we extract the full program and classify it as a test program–this contributes to the"Generated Test Inputs"count. If the output contains no code at all, we discard it and do not count it toward any totals. Hence, for the reported statistics, we consider an output successful if it contains a program, although this does not guarantee that the program compiles. The programs that compile are labelled "Compilation OK" programs. The statistics first indicate the number of programs generated by the LLMs, followed by the number of those that were compile successfully. We further discuss it next.

Throughput Figure 8 shows the throughput of our LLM-based test input generation approaches within 25 hours, grouped by the LLM. The solid black, grey/light-grey stripes and grey/white stripes, represent the total number of generated test inputs, the number of test inputs that were compiled, and the number of non-crashing and non-hanging test inputs, respectively. The bold numbers above each bar represent Y-axis values stated explicitly for clarity. For example (see right end of Fig. 8), in 25 hours **GPT-3.5-turbo (SSBSE 2023)** generated **1869 C parameterised files**: 1086 compiled with `GCC-11-O3`, out of which 823 led to a non-crashing binary, potentially suitable for fuzzing with `AFL++`. These form a valid set of LLM-generated test binaries for `AFL++` (grey/white stripes bar).

In Fig. 8, the largest set is the `GPT-3.5-turbo`, followed by `Magicoder`. `Llama2` and `CodeBooga` come next, with relatively similar population sizes (after

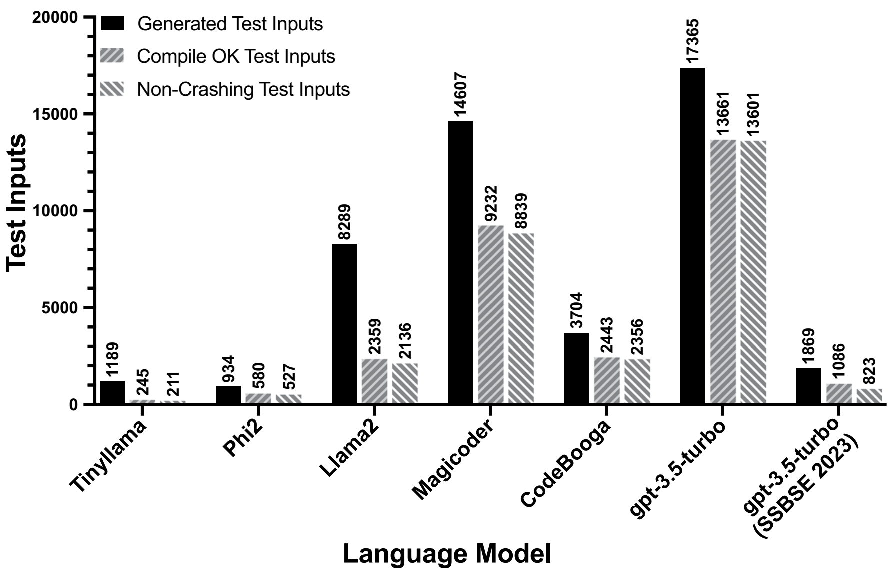

Fig. 8 Number of (unique) `gem5` test inputs generated with different language models during 25 hours. The LLM models are ordered by the number of ANN parameters from tiny to large (by today's standards). Note that GPT3 LLM was run on `OpenAI` 's cloud servers whilst the other LLMs were run locally on our A30 Tensor GPU 80-core server

filtering invalid inputs). This suggests that when model outputs compiled successfully, it was often a good indication that these input tests, used as seeds, ran without crashing or hanging. The SLMs (`TinyLlama` and `Phi2`) generated the smallest sets. The choice of a specific LLM was usually more crucial than its purpose (code or general) or size (e.g. `CodeBooga` vs `Magicoder` in Fig. 8). However, SLMs had significantly lower throughput, suggesting that the model's size affected the generation rate. This is likely because smaller models tend to generate a higher proportion of low-quality, non-code answers, reducing the number of responses that qualified as test inputs; moreover, their generated code was less likely to pass compilation. Notably, `GPT-3.5-turbo` (SSBSE 2023) demonstrated poorer throughput than `GPT-3.5-turbo`, implying that the zero-shot prompting approach was more efficient in terms of throughput.

For `AFL++` fuzzing, only non-crashing and non-hanging instances can be used as seeds due to its limitations (its heuristics are likely to classify these as ignored seeds). However, for testing (RQ1), we used all "Compiled OK Test Inputs" (grey/light-grey stripes) because crashing inputs can still reveal bugs (e.g. the simulator crashes with a successful native run).

Next, we discuss the coverage and testing with LLM-generated inputs pre-fuzzing results, using"Compiled OK Test Inputs".

Coverage We used distinct binaries compiled from the programs of the LLM-generated test inputs for coverage measurements of this part. The line coverage results were as follows: `TinyLlama`: 35964; `Phi2`: 37124; `GPT-3.5-turbo` (SSBSE 2023): 40876; `Llama2`: 41667; `CodeBooga`: 42563; `Magicoder`: 44418; and `GPT-3.5-turbo`: 44781. We used the raw coverage data of each set presented above, for further analysis of the initial corpus in Section 6.2.

`GPT-3.5-turbo` achieved the highest coverage (it also had far higher throughput than any other `Ollama` models). It was expected because the model ran on a much more powerful platform compared to our local GPU machine. The smaller LLMs, `Phi2` and `TinyLlama` had the lowest coverage, which aligned with expectations given their size and capabilities.

`Magicoder`, although smaller in size (3.8 GB), surprisingly outperformed `CodeBooga` (19 GB). This unexpected result highlights `Magicoder` 's efficiency in generating valid and compilable code. The other models performed as anticipated based on their specifications. Yet, notably, all three medium models (`Magicoder`, `CodeBooga`, `Llama2`) achieved higher coverage than `GPT-3.5-turbo` (SSBSE 2023). `GPT-3.5-turbo` (SSBSE 2023)'s reliance on test program sources restricted the programs it could have generated and thus the coverage it achieved.

Bugs We tested cases where the simulation and native-run disagreed deterministically (the same simulation ran twice on the same test input returned the same result, the C program is UB-free, etc), with a 50-second timeout and up to 10 lines of standard output in a native run. This strategy identified bugs where the native run crashed or hung while the simulation ran correctly and vice versa, or mismatches. Cases in which both simulation and native run resulted in a crash or a hang were excluded. Each bug was manually investigated and categorised.

Table 4- *bugs classification* Table 4 presents the bugs identified in our investigation, excluding instances where the native run finished within 50 seconds but the simulation did not, or failures only in native. However, we recorded timeouts as follows: 1 test input each for `TinyLlama` and `Phi2`, and 3, 16 and 10 test inputs for `Llama2`, `CodeBooga`, and `GPT-3.5-turbo`, respectively. Table 4 includes the bug type ("Bug"), the bug description, and columns A to F, which represent the number of test inputs per bug found by each model (**A**: `TinyLlama`, **B**: `Phi2`, **C**: `Llama2`, **D**: `Magicoder`, **E**: `CodeBooga`, and **F**: `GPT-3.5-turbo`). We found no new bugs with `GPT-3.5-turbo` (SSBSE-2023) and hence excluded it from the table.

In total, we found 1 segmentation fault, 6 panic errors, and 14 mismatches, some were related to unimplemented instructions in the simulation, as classified in Table 4. The highest number of bugs were identified by `GPT-3.5-turbo` (17), followed by `CodeBooga` (14) and `Magicoder` (11). `Llama2` (10), `Phi2` (6) and `TinyLlama` (4), identified a lower number of bugs, focusing mainly on panic errors. We reported bugs #7, #16, #17, #18 and #20 in Table 4 to `gem5` issues. Bug #7 involved the incorrect simulation of two numbers' subtraction, likely related to the long double type's implementation in the simulator (see report number #1227). Bugs #16, #17, #18 and #20 were panic error crashes of `gem5` when simulating an invalid binary, that is, instead of indicating a crash in the simulated binary, the whole system crashed (see report numbers: #1507, #1483, #1506 and #1508, respectively. In addition, three bugs were fixed between two versions tested as reported in Table 4 (bugs #13, #14 and #19).

These bugs were reported after several discussions with the developers. They noted that the panic error is particularly interesting (e.g. gem5 developers meeting - August 2024). We consider reporting the remaining bugs after clarifying the developers' requirements, as discussed in Section 7. For example, bug #2 (`system(command)` unimplemented) might be a known issue, and it raises the question of whether to report it as a bug or as an additional test for the `gem5` test suite.

Table 5- *Analysis of bugs' classification* We marked a test as"possibly exposing a bug"if the execution of the program on X86 and the simulator gave different results–for example, if one crashed or froze while the other worked. We applied this strategy to 100% of the test inputs generated by each of the LLMs. From this auto-analysed test input step, we got a population of"possibly exposing a bug"test inputs.

However, not all test inputs that looked like bugs were actual problems – for example, small differences when printing the `PATH` variable or tiny rounding differences in numbers were fine and not treated as bugs. At this stage, we used a semi-manual approach: 1) We first removed the acceptable differences by looking at the outputs. 2) Then, we grouped test inputs by behaviour, mismatches in one set and crashes or panics grouped by their trace (this step was automated). 3) Finally, we looked at each group and manually analysed one example from each: we examined the program's

Table 4 Faults in `gem5` found using test inputs generated by different LLM models

#	Bug	Bug Description	A	B	C	D	E	F
1	Mismatch	Variable's value is random in x86 but fixed in simulation.	-	-	1	2	6	9
2	Unimpl.	`system(command)` unsupported.	1	-	-	1	1	-
3	Mismatch	`wchar` inputs are partially supported in simulation, leading to a mismatch with x86.	-	-	-	2	9	17
4	Mismatch	Time's value is fixed in simulation as: "Sun Sep 9 02:46:40 2001".	-	-	1	1	1	35
5	Mismatch	Bug related to complex numbers. (Bug fixed between the two tested versions.)	-	-	-	-	1	-
6	Mismatch	The simulation does not handle −0.0 correctly. Assign 0.0 instead of −0.0 when obtained via `strtold` with no related warning.	-	-	-	-	-	1
7	Mismatch	Numbers subtraction was simulated wrongly due to long double simulated as an 80-bit float. We reported the bug, GitHub, gem5, #1227.	-	-	-	-	-	7
8	Unimpl.	Fatal error due to syscall clock_nanosleep unimplemented at src/sim/syscall_emul.cc:67.	-	1	7	2	3	1
9	Unimpl.	Fatal error due to unimplemented syscall dup3 at src/sim/syscall_emul.cc:67.	-	-	-	-	-	5
10	Unimpl.	Simulation warning 'fdivr' unimplemented.	-	-	-	-	-	1
11	Unimpl.	Simulation warning 'fcomip' unimplemented.	-	-	-	-	3	4
12	Unimpl.	Simulation warning 'fscale' unimplemented.	-	-	-	1	1	5
13	Unimpl.	Simulation warning 'fscale' unimplemented. Related to `strtold`, led to missimulation. The bug was fixed in version 23.1.0.0 (May 2024).	-	-	-	-	1	1
14	Unimpl.	Simulation warning 'fxam' unimplemented. Related to `atof`, led to missimulation. The bug was fixed in version 23.1.0.0 (May 2024).	-	-	-	-	-	1
15	Panic error	Tried to execute unmapped address at src/arch/x86/faults.cc:166.	-	-	1	1	-	-
16	Panic error	Tried to read unmapped address (same loc.). We reported the bug, GitHub, gem5, #1507.	4	7	54	10	12	53
17	Panic error	Tried to write unmapped address (same loc.). We reported the bug, GitHub, gem5, #1483.	1	2	6	-	1	4
18	Panic error	src/sim/faults.cc:60: panic: panic condition !FullSystem occurred: fault (Divide-Error) We reported the bug, GitHub, gem5, #1506.	-	1	3	2	-	7
19	Panic error	src/sim/faults.cc:60: panic: panic condition !FullSystem occurred: fault (General-Protection). The bug was fixed in version 24.0.0.1 (Aug 2024).	3	3	85	11	14	84
20	Panic error	src/arch/x86/faults.cc:131: panic: Unrecognized/invalid instruction executed. We reported the bug, GitHub, gem5, #1508.	-	1	6	1	4	8

Table 4 (continued)

#	Bug	Bug Description	A	B	C	D	E	F
21	Seg. fault	Assertion violation due to UB in C test code (address and type conversation), at src/sim/fd_array.cc:321.	-	-	6	-	1	-

Columns A to F are the number of instances of each bug found by each model: **A**: `TinyLlama`, **B**: `Phi2`, **C**: `Llama2`, **D**: `Magicoder`, **E**: `CodeBooga`, and **F**: `GPT-3.5-turbo`. A bug can be **(1)** a missimulation (Mismatch), **(2)** an unimplemented functionality message leading to missimulation or a crash (Unimpl.), or **(3)** two types of crashes: panic (Panic error) or segmentation fault (Seg. Fault)

Table 5 Faults in `gem5` found using test inputs generated by different LLM models: number of LLM generated binaries (col 2), number checked by hand (col 4), those exposing bugs (cols 6, 7)

	Auto-Analysed		Manually Analysed		Bugs	
LLM	Test Inputs Considered		Test Inputs	(%)	Test Inputs Exposing a Bug	Unique Bugs
`TinyLlama`	**211**	100%	23	10.9%	9	4
`Phi2`	**527**	100%	23	4.36%	15	6
`Llama2`	**2136**	100%	50	2.34%	170	10
`Magicoder`	**8839**	100%	18	0.20%	47	11
`CodeBooga`	**2356**	100%	66	2.80%	58	14
`GPT-3.5-turbo`	**13601**	100%	187	1.37%	237	17
`GPT-3.5-turbo` (SSBSE 2023)	**823**	100%	2	0.24%	0	0

binaries, crash messages (i.e. the crash trace), and the test input behaviour under small modifications (code or input) or compilation with a different compiler (we used `clang 12.0.0`).

An example of a group of test inputs is shown in Listing 4. In this case, we manually analysed the first test input, reported it as a bug to the developers (#1483), and waited for their feedback. If the bug turned out to be unique or needed more investigation, we analysed the rest of the inputs; otherwise, we considered the group handled and did not review the others. Although this group contained 14 test inputs (in Table 4), we only analysed one – it counts as a single instance of manual analysis, not 14.

```
>> Test /home/ubuntu/experiment-7/CodeBooga/input/test_input_665939085707730.txt
src/arch/x86/faults.cc:166: panic: Tried to write unmapped address 0x7ffff7feff8.
--
>> Test /home/ubuntu/experiment-7/Llama/input/test_input_445878184857718.txt
src/arch/x86/faults.cc:166: panic: Tried to write unmapped address 0xffffffffffffffff.
--
>> Test /home/ubuntu/experiment-7/Llama/input/test_input_461252773633251.txt
src/arch/x86/faults.cc:166: panic: Tried to write unmapped address 0x3fffffff677a.
--
>> Test /home/ubuntu/experiment-7/Llama/input/test_input_462721458102543.txt
src/arch/x86/faults.cc:166: panic: Tried to write unmapped address 0x26000.
--
.....
--
>> Test /home/ubuntu/experiment-7/gpt3.5-new/input/test_input_1707660028.txt
src/arch/x86/faults.cc:166: panic: Tried to write unmapped address 0x10102464c457f.
```

Listing 4 The test inputs' grouped by error message "`panic: Tried to write unmapped address`".

Table 5 presents statistics for test inputs analysed per LLM model (LLM col.). Columns 2-3 show the total number of test inputs per model with 100% of them automatically analysed. Columns 4-5 show how many were manually analysed and their percentage out of the total (e.g. for `TinyLlama`, 23 test inputs is around 10.9% out of 211 test inputs). Column 6 lists how many tests revealed a bug and column 7 reports the number of unique bugs found per model. In total, we manually examined 23 `TinyLlama`, 23 `Phi2`, 50 `Llama2`, 18 `Magicoder`, 66 `CodeBooga`, 187 `GPT-3.5-turbo`, and 2 `GPT-3.5-turbo` (SSBSE 2023) generated test inputs flagged by our strategy (Table 5). An example of a manual inspection process of bugs is given in Section 7.1. The numbers were lower for models that produced programs with a higher rate of non-determinism, but the results generally followed the model's size. The exceptions were `Magicoder`, which exhibited a higher rate of non-determinism in its generated code and `GPT-3.5-turbo` (SSBSE 2023), which depended on test programs as input, limiting its search space as discussed above.

RQ1 Answer. All models found bugs in `gem5`. We found 21 bugs in `gem5`, 14 of which were missimulations or unimplemented functionality in the simulator. The effectiveness of LLM test input generation –measured by coverage, throughput, and bug-finding capability– generally improved as the model size increased.

Factors such as non-determinism in the generated code or restricting the generation to test program sources (which limits the search space) negatively impacted the model's test input generation capabilities. We expect these two factors to also affect the efficiency of fuzzing with `AFL++` (RQ4).

6.2 RQ2: Fuzzing preparation (I) - corpus minimisation

Fuzzing outcomes are heavily influenced by seed minimisation and corpus selection, recommending against input corpora exceeding 100 test inputs (Herrera et al. 2021; AFL 2022). A compact yet diverse corpus enhances fuzzers' (like `AFL++`) ability to uncover new paths and bugs in the SUT. In contrast, a large corpus extends startup time, diminishing actual fuzzing time, and exhausts resources quickly.

All our input corpora had over 100 test inputs. We used `AFL++` 's corpus minimisation analysis to remove redundancy from the LLM generation process. (`AFL++` 's minimisation process employs an approximate edge coverage heuristic, evaluating original test suites and eliminating redundant test inputs, i.e., if they cover transitions already addressed by other test inputs.)

While filtering invalid inputs (RQ1, Fig. 8, grey/white stripes bar) provided some insight into the quality of LLM-based code generation and `AFL++` 's potential seed selection (preferred vs ignored), it was merely a naive estimation. In RQ2, we investigate a more comprehensive approach using `afl-cmin` analysis to enhance `AFL++` 's GA performance, aiming to minimise the corpus based on edge coverage heuristics (ensuring higher diversity) rather than output behaviour.

To generate data to answer RQ2, we used data from RQ1's 7 sets and processed with `afl-cmin`, resulting in 14 corpora (initial and minimised). Table 6 summarises the statistics on the population size and coverage after corpus minimisation using

Table 6 Results of applying `afl-cmin` to the different corpora of test inputs to minimise them

	TinyLlama	Phi2	Llama2	Magicoder	CodeBooga	gpt-3.5-turbo	gpt-3.5-turbo (SSBSE 2023)
Initial Size	211	527	2136	8839	2356	13601	823
Minimised Size	206	366	613	719	612	703	442
Reduction	2.37%	31.7%	71.3%	91.9%	74.0%	94.8%	46.3%
Coverage Loss	0.0%	0.054%	0.689%	0.165%	0.113%	0.181%	0.0%

Initial Size (initial corpus size), Minimised Size (size after minimisation), Reduction (percentage reduction in size of corpus), and Coverage Loss (percentage reduction in test coverage)

Note: the coverage loss computed using and comparing with the raw coverage data in Section 6.1

`afl-cmin` on valid test inputs population (Fig. 8 striped grey/white bars). We measured the line coverage of the sets before and after minimisation.

Table 6 shows the original size of each LLM-based input generation corpus (row 1) and its size after initiation with `afl-cmin` (row 2). Row 3 presents the percentage of the minimisation, and row 4 is the line coverage loss due to minimisation. For example, consider `GPT-3.5-turbo` **(SSBSE 2023)** (last column in Table 6). We used `afl-cmin` on 823 of the `GPT-3.5-turbo` test inputs (row 1), resulting in an optimised corpus size of <u>442</u> test inputs (row 2). That is, shrinking the size of the original test set by <u>46.3%</u> (row 3), while losing no coverage (<u>0.0%</u> of the original line coverage as in Section 6.1; row 4).

After minimisation, all corpora contain fewer than 800 test inputs. Table 6 shows that `Llama2` and `CodeBooga` reduced their test sets to about a quarter of their original corpus size. `Phi2` and `GPT-3.5-turbo` (SSBSE 2023) nearly halved in size. `Magicoder` and `GPT-3.5-turbo` underwent a more than tenfold reduction. Since the reduction in line coverage was negligible (or nonexistent for `TinyLlama` and `GPT-3.5-turbo` SSBSE 2023), this indicates considerable redundancy in the original test sets and that `afl-cmin` heuristics effectively predict coverage while reducing the input corpus size. We further investigated these observations when fuzzing, comparing the performance of the minimised sets and the fuzzed corpus diversity (RQ4).

One exception to the above is the case of `TinyLlama`. The reduction was only by 5 test inputs with no reduction in coverage. This is likely due to its small size. However, according to `afl-cmin` analysis, it could also indicate that `TinyLlama` produced a non-redundant test corpus. Additionally, we observed an issue where our coverage script measured +1 line covered with the minimised set despite identical function coverage. This discrepancy occurred in the memory controller area (line 1179 in `src/mem/dram_interface.cc`, `gem5 version 23.0.0.1--SSBSE Challenge Track`). To rule out concurrency dependencies or non-deterministic factors like the rand() or time() functions' effects in the test inputs, we repeated the coverage measurement five times with both sets (before and after minimisation sets). With each repeat, the anomaly persisted.

> **RQ2 Answer.** The test case redundancy, measured by branch coverage, is significantly higher for medium- and large-sized models (approx. 70%-90%), in comparison to small language models. However, regardless of the model size, the coverage reduction is negligible after minimisation.

6.3 RQ3: Fuzzing preparation (II) - selecting parameters values for `SearchSYS`

For fuzzing with our custom mutators, we had to select the parameters beforehand for `afl_custom_fuzz_count` customised function. We ran fuzz testing for 1 hour, repeating the experiment 30 times per test set using the tiniest corpus for performance (i.e. `TinyLlama`-cmin, Fig. 8), totalling 900 hours of fuzzing. This process aided in identifying the optimal set for fuzzing. Data was collected by fuzzing the smallest corpus with 30 different `SearchSYS` parameter settings files for 1 hour each, repeated 30 times, totalling 900 hours of fuzzing. Data was aggregated by setting (1-30).

Figure 9 illustrates the outcomes of fuzzing with Settings 1 to 30 (numbered 1 to 30 on the x-axis) for 1 hour, averaging all 30 repeats per set. We evaluated four measures: new test inputs in the fuzzed corpus (top-left), crashes found during fuzzing (top-right), new edges covered (bottom-left), and search depth (bottom-right). The edge and depth statistics are part of the edge coverage metrics, which give us (among other metrics available in `AFL++`), the number of covered blocks and the path depth reached during 1 hour of fuzzing. These measures assessed fuzzing effectiveness in test input generation (top) and depth/coverage (bottom). Setting number 27 performed best in the crashes found and search depth. Setting 3 had the highest new edge coverage while Setting 30 generated the most test inputs. Consequently, we set `afl_custom_fuzz_count` to be 17, 84 and 66 for Operator 1, 2 and 3, Section 3.2.1.

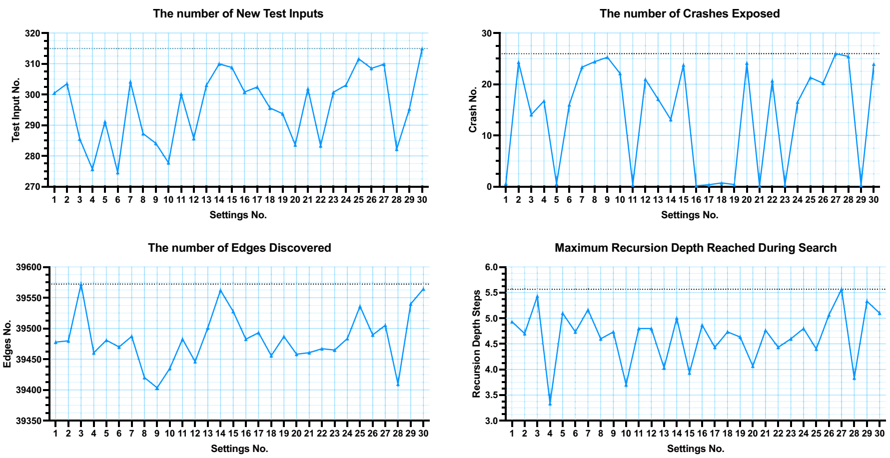

Fig. 9 `AFL++` corpus and edge coverage metrics after 1 hour of fuzzing across the 30 settings. The four graphs present statistics on: the number of new tests (top-left) and crashes (top-right) in the corpus with edges (bottom-left) and the maximal depth (bottom-right) coverage metrics

RQ3 Answer. A 1-hour experiment produced statistics for 30 Settings files (generated randomly), showing significant diversity in the four measures: the number of new test inputs, edges, crashes, and search depth. Given Setting 27 excelled in two measures and performed adequately in others, we employed it as the values of `afl_custom_fuzz_count` for `SearchSYS`.

We consistently applied the same settings file across all experiments in our evaluation for configurations 1 and 3, as the differences between the two configurations did not affect the analysis of the RQ2 statistics above. Configuration 1 included all customisations: type mutator, saving test inputs for later review, and overriding `afl_custom_fuzz_count`. In contrast, Configuration 3 only used the type mutator and overrided `afl_custom_fuzz_count`, without saving test inputs. (See Section 5.3 for the definition of each configuration.) The 30 settings files with the experiment results are available at Dakhama et al. (2024).

6.4 RQ4: Efficiency of fuzzing

We followed the procedure discussed in Section 5.4 to answer the last research question. Specifically, we designed and carried out a set of controlled experiments, fuzzing with each possible combination of input corpus (Section 5.2) and configuration (Section 5.3), using the specification discussed in Section 5.5.

We started 70 instances of Docker with `SearchSYS`, each running for 24 hours with a unique combination of the 14 available input corpora and 5 configurations, totalling 70 instances. After each 24-hour run, we stopped the fuzzing, collected data and repeated the fuzzing process with a new set of 70 dockers. This cycle continued until we completed 10 repeats for each input corpus and configuration combination. Of these 700 runs, 16 failed the coverage measurement stage. These sets were of the fuzzed inputs with `TinyLlama` initial corpus (15) and `GPT-3.5-turbo` (SSBSE 2023) initial corpus (1). All experiments with minimised corpora were completed successfully. Figure 11 gives the averages of the runs which ran to completion. Inspection of the standard error of the means (shown with error bars) suggested that any data gathered from the failing run would not have had much impact and was unlikely to change the conclusions we could draw without it. For example, in Fig. 11 for `Phi2` (top right) the blue and grey lines already lie close to each other (cf. the error bars).

Data was collected from 700 sets generated by fuzzing 14 input corpora (7 initial and 7 minimised) for 24 hours each with 5 configurations, repeating each combination 10 times, totalling 16 800 hours of fuzzing. The data was aggregated by input corpus and configuration into 70 groups, with results presented as averages of the 10 repeats per group. During bug investigations, due to the extensive data requiring semi-manual inspection, we used a single set (arbitrarily we chose the last set, set 10).

Throughput Figure 10 shows the number of new test inputs generated during 24 hours of fuzzing per initial and minimised corpora (14 corpora, in total) across five configurations. The size of the initial corpus is excluded from the total number (i.e., the bars represent the delta between the sizes of `AFL++` 's queue after 24 hours and the input corpus). The grey, grey/light-grey stripes, grey/white stripes, light grey, and blue (dark grey, in greyscale mode) represent the mean values for configurations 1 to 5, respectively. The additional fuzzed test inputs persistently saved when using configurations 1 and 2 are excluded from Fig. 10 due to the numbers being on a very different scale. Moreover, as these inputs were not part of the queue and did not affect the GA component during fuzzing, including them would be misleading.

Yet, saving test inputs to persistent storage (the *Throughput Improvement by Keeping all Fuzzed Inputs* variant) can theoretically significantly increase the rate of generated test inputs for the post-fuzzing differential testing stage. However, by comparing the performance of Configuration 2 and Configuration 3 in Fig. 10, we observed that the final queue size was mostly smaller when saving these test inputs, indicating a performance decrease with this throughput improvement. This was expected because writing to persistent storage introduces overhead from both I/O operations and the management of a larger set of test files, which probably neutralised potential throughput gains. Furthermore, many of these additional test inputs were duplicates, as they did not pass through `AFL++` 's fitness function, contributing little to overall fuzzing efficiency. This further suggests that the two customisations related to overriding `afl_custom_fuzz_count` and independent mutators (to allow better heuristics for `AFL++` 's GI components) probably yielded a more diversified queue than just keeping all test inputs.

The best throughput of new fuzzed test inputs (in the queue) is achieved by `GPT-3.5-turbo`**-cmin (SSBSE 2023)** Configuration 3 (1258.6), `TinyLlama` Configuration 3 (1246.3), `GPT-3.5-turbo` **(SSBSE 2023)** Configuration 1 (1233.2) and `TinyLlama`**-cmin** Configuration 1 (1203.4), all with around 1200 new fuzzed generated test inputs. The lowest rate was achieved mainly by `GPT-3.5-turbo`, which seemed to have a low generation rate in general (all experiments ended with at most 300 new test inputs). More specifically, `GPT-3.5-turbo` Configuration 5 (15.6), `Magicoder` Configuration 5 (24.4), `GPT-3.5-turbo` Configuration 3 (40) and `GPT-3.5-turbo` Configuration 1 (46.1). Considering that the `afl-cmin` and `TinyLlama` sets commonly had the highest throughput rates, while the larger sets appeared to have the lowest rates: these results support the general recommendation to keep the corpus small (Herrera et al. 2021; AFL 2022). Another interesting result was that the performance of `GPT-3.5-turbo` (SSBSE 2023) and `TinyLlama` was roughly the same across all sets, regardless of whether using minimised corpora or not. This indicates that using semi-manually selected high-quality test inputs with a large language model might achieve similar generation rate during fuzzing to a small LLM, locally managed with a fully automated generation approach, due to differing objectives – coverage-directed fuzzing aims to increase coverage, while LLMs prioritise text diversity.

Lastly, Configuration 5 generally showed the lowest performance across all experiments (with the same input corpus), except with `GPT-3.5-turbo`**-min**, where it

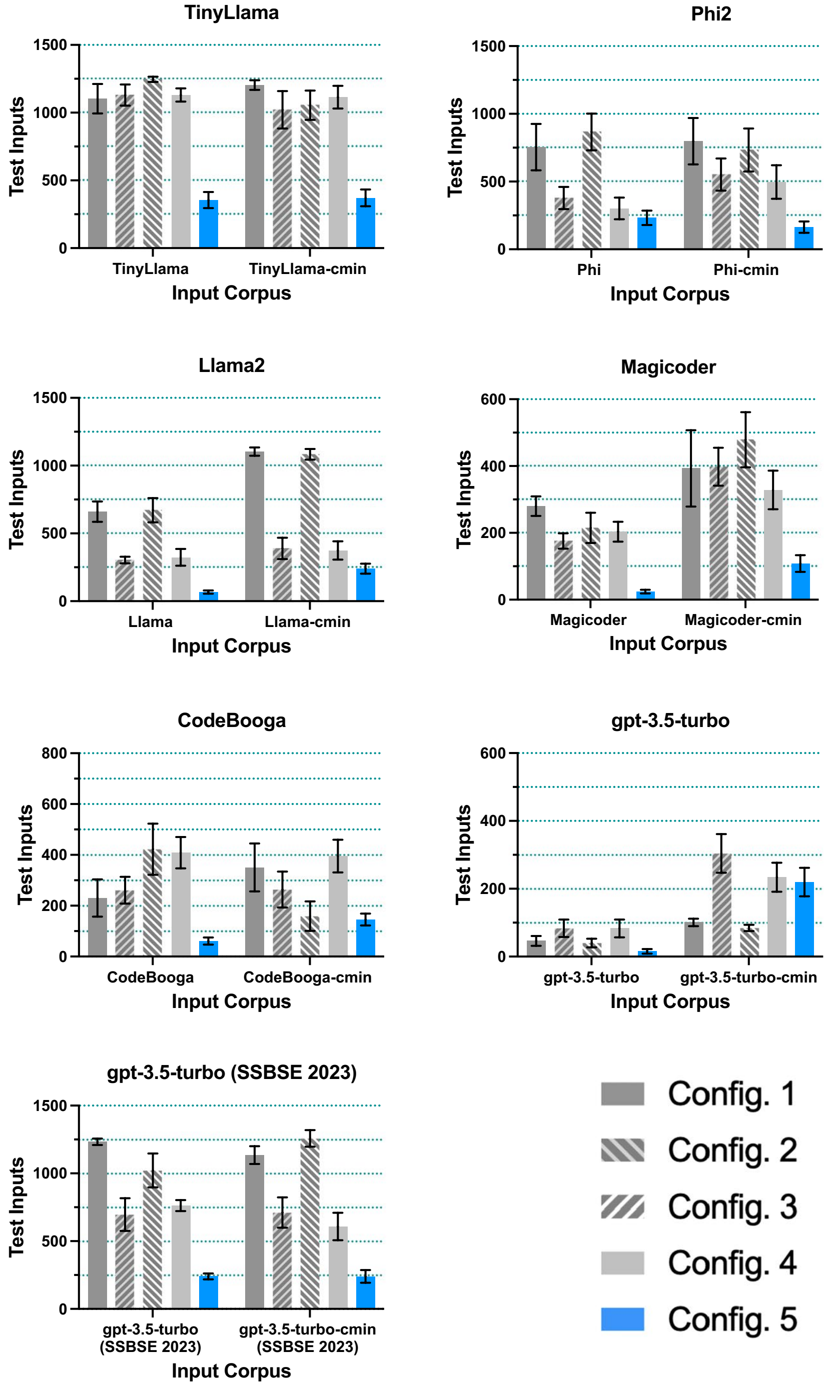

Fig. 10 Mean size of the AFL++ 's queue after 24 hours of fuzzing per LLM model, with each of the five configurations. (The error bars represent the standard error of the means (SEM))

ranked in the middle. This suggests that our additions generally improved throughput. The largest improvement was observed between Configuration 5 (`SearchGEM5` as is) and Configuration 4 (`SearchSYS` with no throughput improvement customisations), indicating that our customisation of the mutators had an important impact.

Coverage Figure 11 shows the average line coverage additions on top of the baseline coverage in Section 6.1. This was done for both initial and minimised corpora (14 in total) across five configurations, with 10 repetitions of fuzzing per corpus and configuration. The dashed light-blue line is the line coverage achieved at the end of fuzzing with the initial corpora, while the solid grey line shows the results with the minimised corpora. The line coverage of the initial and minimised corpora before fuzzing (i.e. the baseline Section 6.1) is marked by the dashed black and dotted blue lines, respectively.

The highest coverage is achieved by each LLM source:

- `Magicoder` Configuration 3, minimised corpus, 45 134.7 lines
- `GPT-3.5-turbo` Configuration 2, with no effect regarding initial or minimised corpus, hinting that `SearchSYS` could little diversify the test inputs at that stage, 44 959 lines
- `Llama2` Configuration 3, minimised corpus, 44 020.7 lines
- `CodeBooga` Configuration 4 with the initial corpus, 43 303.5 lines. Configuration 1 might possibly be better occasionally with the initial corpus but due to variations in the measurements this cannot be determined
- `GPT-3.5-turbo` (SSBSE 2023) Configuration 1, initial corpus, 43 187.5 lines

At a much lower scale, we have the small models, below 40 000 lines covered, with `Phi2` (Configuration 1, initial corpus, 39 001.5) and last `TinyLlama` (Configuration 3, initial corpus, 38 498.2 lines).

From these results, we observed that the baseline coverage achieved by the initial or the minimised corpus can predict to some extent the coverage after fuzzing (e.g. `Magicoder` outperformed `GPT-3.5-turbo` even though `Magicoder` 's initial and minimised corpora had lower baseline coverage). Commonly, Configuration 3 performed the best, Configuration 5 was outperformed by the others, and using the minimised corpus led to better performances when the initial corpus was large. Lastly, `GPT-3.5-turbo` (SSBSE 2023) with Configurations 1 and 3 diversified the test inputs during fuzzing similarly to the smaller models, `Llama2`, `Phi2`, and `TinyLlama`, suggesting that a complex few-shot approach could be replaced with zero-shot prompting on smaller models in the context of fuzzing. However, `SearchSYS` likely diversified test inputs more effectively for `Phi2` and `TinyLlama` due to their much lower baseline coverage.

Bugs We manually inspected each fuzzed test input that led to a performance bug or a mismatch.

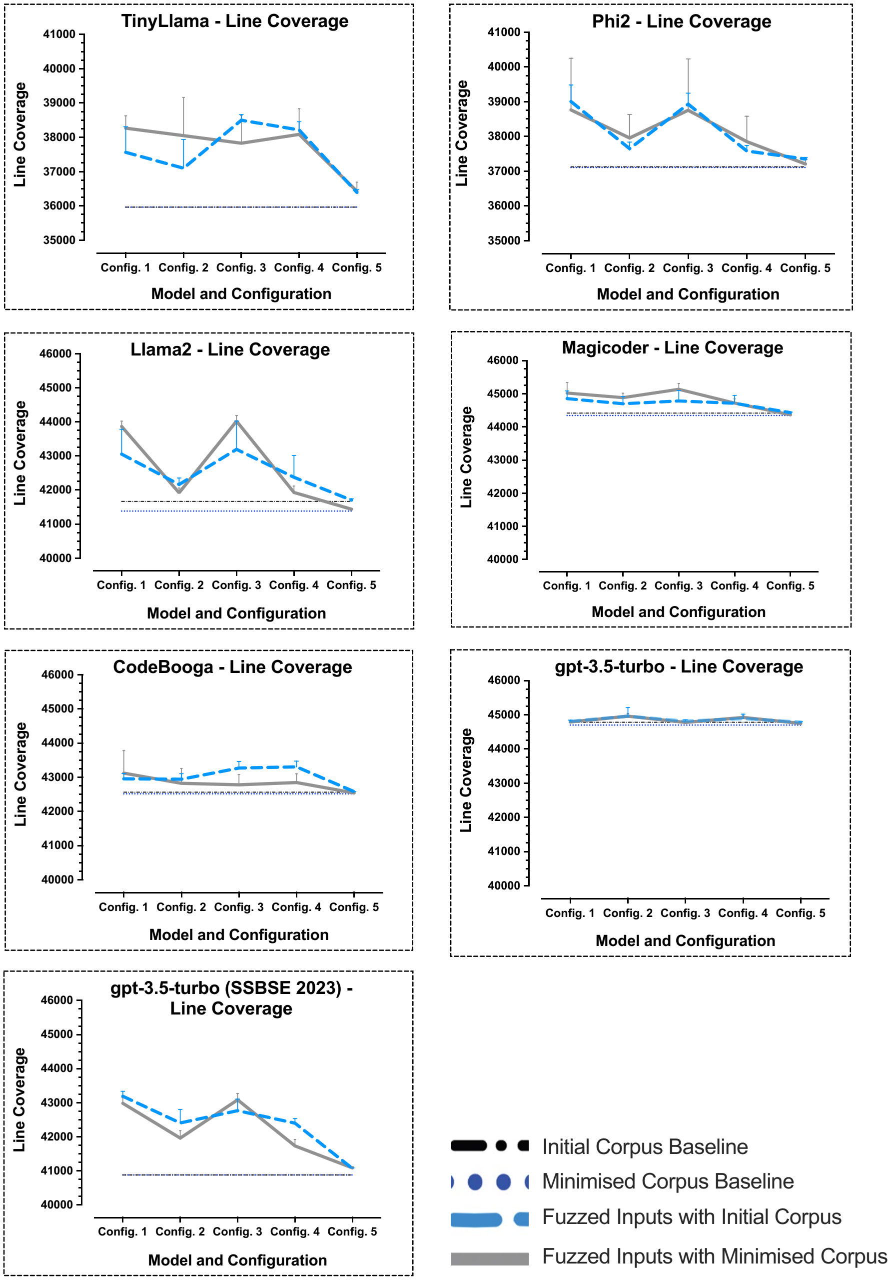

Fig. 11 Aggregation of 700 sets as the mean line coverage with the `AFL++` 's queue after 24 hours of fuzzing per LLM model with each of the five configurations. Notes: (1) Aggregation is on top of the baseline coverage of the initial and minimised corpora. (2) The line T error bars give the standard error of the means (SEM). (3) `TinyLlama` and `Phi2` scale much lower than the others. Consequently, their y-axis scale is set between 35 000 and 41 000, instead of 40 000 to 46 000

Performance bugs include timeouts and out-of-memory instances. *Mismatch bugs* include cases where the simulation ended normally but the native run failed (i.e. bug masking), the simulation crashed with the native run exiting normally, or both ended successfully but produced different outputs (e.g. printed different results or had different return code).

During this inspection, we manually classified some bugs as bugs #3, #8, #9, #15 and #16 from Table 4. Additionally, we identified 9 unique potential crashes. Due to the large number of bug instances found and the absence of source code for most fuzzed test inputs, requiring time-consuming manual decompilation, we did not further classify the remaining bugs. However, we manually inspected a mismatch bug in Section 7. We decompiled the fuzzed test input binary executable for manual inspection and tried to reduce the bug.

Table 7 presents the number of fuzzed test input instances exposing bugs in `gem5` found during 24-hour fuzzing. The columns "Performance", "Bug masking", "Crash", and "Diff. output" in the table are the total number of fuzzed inputs that resulted in performance bugs (timeouts or out-of-memory errors), bug masking, `gem5` crashes, and output mismatches, respectively. These counters are aggregated by configuration (as described in Section 5.3) using data from all 14 input corpora.

The highest number of 62 148 issues in total was found by `SearchSYS` (Configuration 1), followed by Configuration 2 (34 273), Configuration 3 (2 954) and Configuration 4 (1 772). `SearchGEM5` (Configuration 5) had the poorest bug ability discovery, with only 295 instances, in total. Per category of bugs: The highest number of bug masking and crashes were found by Configuration 1, while performance bugs and mismatched output triggered the most by Configuration 2. To sum,

- The bug discovery ability improves when fuzzing with our customisation.
- The *Throughput Improvement by Keeping All Fuzzed Inputs* customisation has a greater impact than optimising `afl_custom_fuzz_count`.

This is because Configuration 2, which only keeps all fuzzed inputs to improve throughput, outperformed the two others (Configuration 3 uses solely the optimising `afl_custom_fuzz_count` option, while Configuration 4 includes neither of the throughput improvement). We shared our analysis as an Excel file, with additional details on the bugs we classified so far, in Dakhama et al. (2024).

Table 7 Number of fuzzed test inputs exposing bugs in `gem5` found during 24-hour fuzzing aggregated by configuration

	Performance	Bug masking	Crash	Diff. output	Total
Conf. 1 (`SearchSYS`)	458	**54642**	**2654**	4394	**62148**
Conf. 2 (First Imp.)	**908**	26360	2261	**4744**	34273
Conf. 3 (Second Imp.)	106	2531	82	235	2954
Conf. 4 (No Imp.)	40	1574	91	67	1772
Conf. 5 (`SearchGEM5`)	31	185	21	58	295
Total	1543	85292	5109	9498	
Friedman (p-value)	▼0.0737	▲▲4.53e−05	▲0.0261	▲0.0186	

Choosing the best configuration: Regarding the different configurations, we applied the Friedman test and the corresponding post-hoc tests to identify statistically significant differences across each metric and configuration (with a p-value threshold of 0.05). According to the general test, there is a statistically significant difference across all configurations for every metric except for performance, which has a p-value higher than 0.05. Moreover, bug masking shows a strong difference across the distribution of results for the various configurations.

We employed the Nemenyi-Friedman post-hoc tests to perform pairwise comparisons between the configurations. In terms of crashes, we found statistically significant differences between configuration 5 (`SearchGEM5`) and 1 (`SearchSYS`), configuration 5 and 2 (First Imp.), configuration 3 (Second Imp.) and 1, and configuration 4 (No Imp.) and 1. This indicates that the improvements introduced in `SearchSYS` are significantly better for detecting crashes compared to all other configurations, except for configuration 2 (First Imp.), which shows similar performance.

For differential testing, there is a statistically significant difference between configurations 5 and 1, demonstrating that the new methodology significantly improves upon our previous approach.

With respect to missimulations or bug masking, we observed statistically significant differences between configuration 5 and all others (4, 3, 2, and 1), indicating that the previous method is significantly worse than any of the improved configurations.

Finally, in terms of performance, there is a statistically significant difference only between configurations 2 and 5, and 2 and 4, showing that the First Imp. configuration achieves significantly better performance than the old configuration and generally provides improved results for this metric.

Choosing the best LLMs: With respect to the results of the large language models, Table 8 extends Table 7 by dividing the results according to each language model. This highlights which initial test suite yields the most effective results across the different metrics. The performance of the various language models varies significantly

Table 8 Number of fuzzed test inputs exposing bugs in `gem5` found during 24-hour fuzzing aggregated by configuration (Input-corpus ordered alphabetically)

Input-corpus	Performance	Bug masking	Crash	Diff. output
CodeBooga-cmin	5.8 ± 4.8	38.6 ± 35.6	2.6 ± 3.2	2.4 ± 2.6
CodeBooga	70.6 ± 151.2	35.0 ± 31.8	18.0 ± 22.1	0.0 ± 0.0
Llama-cmin	29.6 ± 39.1	1030.6 ± 2070.9	53.6 ± 111.0	259.2 ± 546.9
Llama	30.4 ± 40.1	915.8 ± 1935.5	77.0 ± 133.7	260.8 ± 335.9
Magicoder-cmin	**104.4 ± 192.0**	836.8 ± 1800.3	39.8 ± 79.0	268.0 ± 457.7
Magicoder	7.4 ± 12.9	109.2 ± 193.1	19.6 ± 41.6	29.2 ± 58.8
Phi-cmin	6.6 ± 9.8	1575.2 ± 3371.8	42.4 ± 55.1	128.4 ± 212.7
Phi	2.0 ± 3.1	398.8 ± 638.1	3.2 ± 4.6	3.6 ± 7.5
TinyLlama-cmin	5.4 ± 5.9	3379.6 ± 4240.6	107.6 ± 211.3	259.8 ± 440.1
TinyLlama	6.2 ± 8.6	**4062 ± 5611**	121.8 ± 169.5	76.0 ± 121.0
gpt3.5-new-cmin	3.6 ± 4.8	59.4 ± 71.0	16.2 ± 18.7	**530 ± 1148**
gpt3.5-new	0.4 ± 0.9	2.0 ± 1.2	115.2 ± 209.1	0.0 ± 0.0
gpt3.5-old-cmin	23.8 ± 36.9	1908.2 ± 3225.8	150.0 ± 255.9	32.0 ± 44.0
gpt3.5-old	12.4 ± 9.8	2706.8 ± 4221.0	**254.8 ± 499.7**	50.6 ± 62.5
Friedman (p-value)	▲0.0042	▲▲2.84e−06	▲▲9.33e−06	▲0.00135

and is generally unstable, as indicated by their standard deviations. This instability suggests that the performance of LLMs is highly variable –either extremely good or poor– across different executions, with no model showing consistent results.

For the metric of discovering performance bugs, the best test suite is the one generated by `Magicoder`-cmin, which discovers an average of 104 bugs. For bug masking, the most effective model is `TinyLlama`, exposing 4062 bugs. In terms of crashes, the best performer is the old version of the test suite generated by `GPT-3.5-turbo` (SSBSE 2023), which detects 255 bugs. Finally, for differential testing, the best result comes from the new version, `GPT-3.5-turbo`-cmin, which discovers 530 bugs. However, it is important to note that none of these results were stable in terms of standard deviation.

After applying the Friedman test to the different LLMs, the general test indicates a clear imbalance across the LLMs, as it passes for all metrics and shows strong statistical significance in bug masking and crashes. However, when applying the Nemenyi-Friedman post-hoc tests, we find that in three out of the four metrics – specifically performance, crashes, and differential outputs – there is no statistically significant pairwise difference between the LLMs. This is likely because the test is performed across all configurations, and LLM performance varies depending on the chosen configuration.

It is important to consider that the Friedman test is a global test and is sensitive to overall variations, thus detecting general imbalances. In contrast, the Nemenyi-Friedman test performs pairwise comparisons, is more conservative, and includes corrections, making it harder to detect significant differences between individual pairs – especially with small sample sizes.

In conclusion, the Friedman test provides evidence that not all models perform equally. However, according to the post-hoc test, no single LLM stands out as significantly better than the others. Their performance depends on the configurations, with configuration 1 being the most suitable for the majority of the metrics.

RQ4 Answer. Configuration 1 (`SearchSYS`) and Configuration 3 (with "Optimising `afl_custom_fuzz_count`") commonly achieved the best code coverage and fuzzing throughput. The best bug-finding capability was observed with Configuration 1 (`SearchSYS`) and Configuration 2 (with "Keeping All Fuzzed Inputs"). Therefore, Configuration 1 is the most effective overall, as it combines both customisations in addition to the mutator customisation (that all but `SearchGEM5` includes), making all three customisations essential for enhancing `SearchSYS`'s bug-finding, throughput and coverage effectiveness.

7 Discussion

In this section, we thoroughly examine two bugs we discovered –a hang and a missimulation– providing a detailed investigation in Section 7.1. Further discussion on bug classification, including crashes and missimulations, can be found at https://youtu.be/hEyhXJg-rbU. We then discuss the bug reports and the acknowledgement of the results in Section 7.2. Lastly, we explore the effectiveness of our two approaches in `GPT-3.5-turbo` generation in Section 7.3.

7.1 Investigation of two gem5 bugs

A performance bug SearchSYS generated several hangs (see Sections 6.1, 6.4 and Dakhama et al. 2023). We did a short investigation using one of these hangs to better understand hang bugs' relevance to the quality of system simulators and to gather developers' feedback.

The program contains a loop in function `main` that calls `fn1` function from that loop. We compiled the program binaries using `GCC-11 -O3` and `Clang-16 -O3`, allowing us to compare the simulation and native runs with two different binaries of the same program. The full bug report and the code are available.

The hang occurred exclusively in gem5 simulation with input values extremely near `INT_MIN` (e.g.−2147483648, −2147483647 and −2147483640). With slightly bigger values, the gem5 simulation finished within a second (e.g.−214748364). These inputs did not cause a hang in a native execution using the same binary. The gem5 developers confirmed the bug but closed this issue, noting that the simulation completes, if allowed to run overnight. This implies that hangs and performance issues are likely lower prioritised in such systems.

A missimulation bug SearchSYS has the ability to generate binaries that make gem5 crash. The system, in such a case, provides some details for diagnosis. For missimulation, we use differential testing, comparing gem5 and native and using the test program for diagnosis.

Fuzzed corpus test inputs leading to mismatches are more challenging to analyse. In the case of a mismatch from the fuzzed corpus, we only know that the results are inconsistent, but without the binary's source code, debugging becomes difficult. Investigating the nature of the bug requires a reverse engineering process to understand which parts of the original binary were modified during fuzzing and how they can affect the system. Apart from this, we need to provide a minimal example that activates the bug.

Here, we discuss our semi-manual analysis of bug #15 (Dakhama et al. 2023) and the challenges involved. In this case, the response of gem5 and the system's response were different for the same binary program and arguments' input. The mismatch detected was as follows; running fuzzed program `.fuzzed.o 0 0 0` in native, it printed: `b.a = 0, c = 22091, x = 0`, while in gem5: `b.a = 0, c = 0, x = 0`.

Figure 12 shows the code of the fuzzed program. Using `radare2` combined with Ghidra's decompiler, we reconstructed the original source code and compared it with the "seed" program that generated this variant. The mutant code is in the normal font style while the original is in comments only when there are differences.

In the mutation process, SearchSYS changed multiple types during the variable definition (lines 3 to 9). This allocated different memory sizes. It also changed the way the functions were called, instead of using `strtol`, as in the original program, it used `atoi` with a similar purpose of transforming the inputs from strings to numbers (lines 14, 19 and 23). It changed multiple elements of pointer arithmetic (lines 13 to 43). Most relevant to the output relates to the changes in lines 37, 40 and 43,

```
ulong main(int32_t param_1, ulong *param_2) {
  uint32_t uVar1;
  unit     uVar2;                      //uint32_t uVar2;
  unit     uVar3;                      //uint32_t uVar3;
  ulong    uVar4;
  uchar    *puVar5;                    //uint32_t uVar5;
  uchar    *puVar6;
  uchar    *puVar7;
  uint32_t  uVar8;                     //uchar *puVar8;
  ulong    uStack_20;

  if (param_1 == 4) {
    uStack_20 = 0x10bb;                //uStack_20 = 0x10c0;
    uVar1 = sym.imp.atoi(param_2[1]);  //uVar1 = sym.imp.strtol(param_2[1],0,10);
    uVar4 = param_2[2];
    puVar5 = &stack0xffffffffffffffe8;  //puVar6 = &stack0xffffffffffffffe8;
    *(&stack0xffffffffffffffe8 + -8) = 0x10c8;
                                       //*(&stack0xffffffffffffffe8 + -8) = 0x10d3;
    uVar2 = sym.imp.atoi(uVar4);       //uVar2 = sym.imp.strtol(uVar4, 0, 10);
    uVar4 = param_2[-0xd];             //uVar4 = param_2[3];
    puVar6 = puVar5;                   //puVar7 = puVar6;
    *(puVar5 + -8) = 0x10d6;           //*(puVar6 + -8) = 0x10e6;
    uVar3 sym.imp.atoi(uVar4);         //uVar3 = sym.imp.strtol(uVar4, 0, 10);
                                       //_obj.c = uVar2;
    uVar8 = uVar1 & 0x1ffffff;         // uVar5 = uVar1 & 0x1ffffff;
    _obj.c = uVar2;                    //
    *0x4024 = *0x4024 & 0x1fff | uVar1 << 0xd;
    uVar3 = uVar3;                     //uVar1 = uVar2;
    if (8 < uVar8) {                   //if (uVarS < 9) {
        uVar3 = uVar2;                 //uVar1 = uVar3;
    }
    *0x4028 = *0x4028 & 0xc0 | uVar8 >> 0x13 & 0x1fff;
                                       //*0x4028 *0x4028 & 0xc0 | uVarS >> 0x13 & 0x:
    _obj.x = uVar3;                    //_obj.x = uVar1 & 0xfffffff;
    puVar7 = puVar6;                   //puVar8 = puVar7;
    *(puVar6 + -8) = 0x113d;           //*(puVar7 + -8) = 0x114a;
    sym.imp.__printf_chk(1, "b.a = %u\n", uVar8);
                                       //sym.imp._printf_chk(1,"b.a = %u\n");
    *(puVar7 + -8) = 0x1156;           //*(puVar8 + -8) = 0x1163;
    sym.imp.__printf_chk(1, "c = %i\n", *0x400c);
                                       //sym.imp._printf_chk(1,"c = %i\n",_obj.c);
    *(puVar7 + -8) = 0x116f;           //*(puVar8 + -8) = 0x117c;
    sym.imp.__printf_chk(1, "x = %i\n", _obj.x);

    uVar4 = 0;
  } else {
    uVar4 = *param_2;
    *(*0x20 + -0x20) = 0x10a6;
    sym.imp.__printf_chk(q, "Usage: %s <val_1> <val_2> <val_3>\n", uVar4);
    uVar4 = 1;
  }
  return uVar4;
}
```

Fig. 12 Decompilation of differential testing bug

where it prints the values. In the original program it printed the structure, while, in the mutant, it printed a variable and a specific memory address.

Analysing the outcome of the program, we can see that the second print differs. In `gem5` with `--isa X86`, it prints 0 while in a `x86` Ubuntu host, it prints garbage (e.g. 22 091). In this case, the expected result is garbage. Multiple runs show the same behaviour. This means that the way systems organise memory is different to `gem5`.

Figure 13 shows a minimised program that we believe exhibits the same issue. However, running this program in both `gem5` and the host environment results in the same outcome: a crash. Decompilation is not expected to produce valid code. To effectively recreate the source for our mutated binaries and minimise them, new methods are required, such as using LLMs to assist in decompilation and minimisation.

```c
#include<stdio.h>

int main() {
  __printf_chk(1,"c= %i\n", *((int *)0x400c));
  return 0;
}
```

Fig. 13 Bug minimization for the gem5 bug in Fig. 12

7.2 Bugs reported and acknowledged

During the development of gem5, we aimed to understand the relevance of the bugs that we identified. We decided to communicate with different stakeholders involved with gem5 's ecosystem and discuss the importance of the discovered bugs. Even though our target architecture is x86, we discussed several of the tool features and discovered bugs with ARM, which was interested in those bugs that were related to mismatches. During our discussions with gem5 developers, they were mainly interested in the panic errors, which are the ones where the simulator crashes. They asked us to report every panic error. These communications have been carried out by email, although we presented several of the bugs discovered by SearchSYS during one of the monthly gem5 developers meetings. Following the meeting, we reported 4 panic errors to the gem5 issue tracker, discovered with LLM-generated test inputs as bug reports: #1483, #1506#1507 and #1508.

7.3 Comparison of manual vs template prompt GPT-3.5-turbo approaches

We compared two approaches for generating test cases using GPT-3.5-turbo: a *semi-manual approach* originally used on an older version of GPT-3.5-turbo (August 2023) and *the template prompt approach* (Section 3.1.2). The original method involved a more manual process, providing extensive context to GPT-3.5-turbo for each test generation request as well as a window of previous tests which served as examples. This method was slower, less efficient, and resulted in fewer test cases being generated. Additionally, the quality of the generated test cases was lower, with less code compiling successfully and achieving lower coverage.

In contrast, the newer approach uses template prompt and Table 1's tokens to accelerate the process. Instead of providing extensive context, we use specific, zero-shot instructions to guide GPT-3.5-turbo in generating the desired code. This resulted in a much higher throughput and quality of code produced, with higher coverage. We found this approach, compared to the original method, resulted in a higher throughput on the current GPT-3.5-turbo version (February 2024).

The improvement in the newer method can be attributed to several factors. Firstly, the template prompt approach provides more specific and direct guidance to GPT-3.5-turbo, reducing ambiguity and improving the relevance of the generated code. Secondly, the newer versions of GPT-3.5-turbo used in this approach may have improved capabilities and performance, contributing to the higher quality of the output. However, due to the closed nature of the GPT-3.5-turbo model, we cannot definitively isolate the impact of the model version from the changes in our methodology. Nonetheless, the overall enhancement in test case generation is evident, demonstrating the effectiveness of the template prompt approach in leveraging `GPT-3.5-turbo` for software testing.

8 Threats to validity

8.1 Internal validity

Comparing against other approaches To the best of our knowledge, there is no other fuzzer specifically designed for system simulator fuzzing while capable of performing differential testing. This lack of comparable alternatives limits our ability to benchmark against other methods. `AFL++` has fuzzing binaries options. We briefly discuss these, arguing their irrelevance in the context of this paper.

The binary-only fuzzing option is relevant when the source code of the SUT or the target is unavailable (AFL++ fork of QEMU). This is an orthogonal approach dealing with the target itself. While this provides flexibility for fuzzing targets without their source code, it does not focus on fuzzing binary test inputs to the SUT.

`AFL++` 's mutators (e.g. bit-flip) can be applied on binary executable test inputs. However, randomly flipping bits in binary executables often results in corrupted binaries that fail to execute properly in both native and simulated environments, leading to many false positives. Moreover, with no access to native `x86` oracle during fuzzing, this option can reduce the efficiency of using GI and coverage, as most mutants result in a crash. Hence, more sophisticated techniques beyond simple bit flips are necessary when fuzzing system simulators.

LLM models Our study included six different models running locally or remotely. Initially, we excluded `StarCoder` (Li et al. 2023) due to its relatively poor performance, which was its most recent version when we started our evaluation. However, the newer version, `StarCoder2` (Lozhkov et al. 2024), appears to be more effective and may be considered in future work.

Seed selection Our work leverages LLMs to automate the generation of test inputs and uses `afl-cmin` for corpus minimisation, addressing the lack of systematic methods for generating seeds in system simulator testing. Without the LLMs, users are required to supply an initial corpus. This challenge leads to a dependency on LLM's code-generation ability. Establishing such a corpus without LLMs inevitably delves into the broader and well-studied issue of seed selection in fuzzing (Herrera et al. 2021), which is not the scope of this work. Our LLM-based approach is a first step toward the generation of benchmarks for simulators' ISA components, enabling future baselines.

Instrumentation Our methodology relies heavily on the correct instrumentation of the simulator's source code, where `AFL++` is used. This may impact the runtime performance of `SearchSYS` and the reproducibility of detected bugs during fuzzing, which may no longer triggered using a non-instrumented simulator. This issue is not unique to `SearchSYS` and is common when using instrumentation. To address it, we automatically tested the test inputs from the fuzzed corpus, using a non-instrumented build of the simulator and cross-checked it with the results from native execution. Furthermore, we manually analysed bug reports generated by SearchSYS before reporting these to the developers.

Fuzzing We used the fuzzing methodology described in Klees et al. (2018), where each experiment lasted for 24 hours. We repeated each experiment –fuzzing each combination of test input corpus and configuration– and reported results that are averages across these repeated experiments. Due to the challenge of maintaining identical experimental environments for each repetition, we repeat each experiment 10 times, which is fewer than the recommended number of 30 repeats to mitigate the randomness inherent in fuzzing (Klees et al. 2018). Yet, Fig. 9 shows that the standard deviation is largely dependent on the LLM used rather than the selected SearchSYS 's configuration (e.g. Phi2 is noisier than TinyLlama). It suggests that the variability in our results is more influenced by the inherent randomness of LLMs rather than fuzzing, making the choice of an LLM model significant. Alongside the results presented here, we also provide the source code used to obtain these results, enabling the reproduction of our experiments. However, system simulators, when used in conjunction with fuzzing, are extremely resource-intensive, particularly in terms of system memory. Therefore, a sufficiently capable machine is necessary to replicate our findings.

Coverage measurement The resources required for using SearchSYS and gem5 make it hard to measure coverage. During our experimentation, we encountered several issues connected with these limitations especially connected with the inputs that TinyLlama generated. Some of these inputs crashed the whole fuzzing process affecting AFL++. This was predominant at employing the fuzzed-by-proxy strategy. As a consequence, the coverage results for TinyLlama contained fewer than 10 repetitions.

8.2 External validity

Hallucinations of LLMs pose a threat by misunderstanding the prompts provided (Perković et al. 2024). This may lead to the generation of invalid programs, programs with undefined behaviours (ISO C, Working Group SC22/WG14 2018) (e.g. uninitialised local variables), or programs with no input. During test input generation, we filtered out invalid programs by compiling them and removing any sources that failed to compile from the input corpus. Programs with undefined behaviours present a genuine risk to valid detection of missimulations. We addressed this issue by analysing mismatches, once a bug is found, to identify and exclude programs with undefined behaviours from the reported missimulation bugs (see Section 8.1).

8.3 Transferability and reproducibility

Although this paper uses gem5 as a use case, AFL++ has been widely applied to various targets (Groce et al. 2022; AFL compiler fuzzer n.d.; Kersten et al. 2017; Pham et al. 2020; Wilk n.d.). Therefore, applying SearchSYS to a different target SUT should be straightforward. Our custom AFL++ mutator is reusable since it performs target-independent mutations.

GPT-3.5-turbo has been trained on a wide range of programming languages, including C, Python and Java. By adjusting the LLM to target the desired programming language and modifying the prompt template tokens in Table 1 accordingly, our method can be easily adapted to generate test inputs for other programming languages (Biswas 2023; Destefanis et al. 2023). However, GPT-3.5-turbo is not open-source with various terms and policies restricting its usage, making our results potentially non-transferable in some contexts with GPT-3.5-turbo model. Ollama is preferable as it allows us to use the exact model locally, ensuring transparency, transferability and reproducibility with full control over the model, its version, and its seed. Similarly, with the same seed, AFL++ and our post-processing steps are fully reproducible and deterministic. Furthermore, fuzzed binaries are not platform-independent – they can only be used on similar operating systems and environments, such as those with the same architecture (e.g. x86) and compatible GLib versions. This requires repeating the fuzzing phase per target architecture.

To effectively use SearchSYS in real-world scenarios, we recommend leveraging local LLMs via platforms like Ollama to mitigate GPT-3.5-turbo's drawbacks discussed above. Our evaluation found Magicoder and Llama2 to offer a strong balance between seed quality (for pre-fuzzing testing) and fuzzing performance. Nonetheless, as LLMs evolve rapidly, we advise re-evaluating the latest versions before launching a long fuzzing campaign. To simplify this process, we provide Docker images and full instructions (Dakhama et al. 2024). Specifically, we suggest: selecting recent LLM versions, generating and minimising initial corpora, tuning SearchSYS parameters (as in RQ3), conducting 24-hour fuzzing runs per corpus, and performing differential testing on initial and fuzzed corpora.

9 Related work

With the emergence of Large Language Models (LLMs) since the Fall of 2022, several researchers have used them to automate software engineering tasks. There are several surveys available that were posted on https://www.arxiv.org in the second half of 2023 (Zhang et al. 2023; Wang et al. 2023; Fan et al. 2023). Of these, Zhang et al. (2023) provide a good summary of recent work that uses LLMs for fuzz testing (see Section 4.3.7 Zhang et al. 2023). Most of the work was proposed in 2023 and uses GPT-based LLMs and user interfaces, particularly ChatGPT.

Among the work closest to SearchSYS, is CHATFUZZ, a tool introduced by Hu et al. (2023) to improve grey-box fuzzing. It automatically prompts ChatGPT to generate seeds for fuzz testing which are similar to existing seeds but that fit better to a given format. Xia et al. proposed Fuzz4All (Xia et al. 2024) which first generates prompts which produce example code snippets for fuzz tester input. These samples are then evaluated and those prompts that generate the largest number of valid inputs are taken to generate more inputs for a given fuzzer. Additionally, Ackerman and Cybenko (2023) use LLMs to generate inputs from natural language specifications, then use LLM to further mutate those, and then, in turn, input these as seeds to a fuzzer.

In the context of mutation testing (DeMillo et al. 1978; Dakhel et al. 2023) ask an LLM to generate initial unit tests, run mutation testing to evaluate them, then enhance the LLM prompt with information about surviving mutants to produce further test inputs. On the other hand, Lemieux et al. (2023) introduce CodaMosa, a search-based software testing tool that prompts an LLM for new inputs during the search process, in cases where the search gets stuck trying to cover a particular callable instruction.

There's little work on fuzz testing hardware simulation software, most are concerned with traditional software or embedded systems (Eisele et al. 2022). Nevertheless, Martignoni et al. (2013) propose a prototype fuzzing tool for CPU emulators, called EmuFuzzer. They used it to test five emulators (QEMU, Valgrind, Pin, BOCHS, and JPC) finding bugs in each of them. Jiang et al. (2022) test three CPU simulators (QEMU, Unicorn, and Angr) for ARM devices. They generate valid inputs which are then automatically mutated using rules that result in syntactically-valid ARM instructions. To generate more tests, they extract constraints which influence the execution path and use a symbolic execution engine. Yu et al. (2016) propose an automated framework, called VDTest, to test virtual devices within full system emulators. They extract test templates using static analysis and subsequently employ combinatorial testing (Petke 2015) to generate new test inputs. They tested eleven virtual devices, including one from `gem5`, revealing 64 faults. Similarly to our approach, they use differential testing between real devices and emulators to discover inconsistencies.

Our approach uses the `AFL++` fuzzer in an unconventional way, replacing test input fuzzing with a more sophisticated input format as discussed in Section 3, an idea that was adopted by various domains such as compiler testing (Groce et al. 2022; Aschermann et al. 2019) or network protocol analysis to manipulate network protocols for fuzzing (Pham et al. 2020).

In terms of testing `gem5`, while there are efforts in verifying its architectural compliance (Bruns et al. 2021) and the integrity of its code, `gem5` testing or verification approaches (Bossuet et al. 2023) that can go deeper into its internals, such as `SearchSYS`, are essential to validate its many possible options.

Serebryany et al. (2021) present a hardware fault diagnosis tool in use in Google data centres whose tests cases were derived with help from software fuzzing of CPU simulators similar to `gem5`, but they only mention `gem5` as a possible future tool they would like to use. Rajeev and Song (2023) use `gem5` to test their fuzz inputs rather than fuzzing `gem5` itself.

Thus far only our previous work (Dakhama et al. 2023) used fuzzing to test `gem5`. In this preliminary version of our work, we integrated LLMs and SBSE for the purpose of testing system simulators, and developed a prototype of `SearchSYS`. Here we extend our previous approach (Dakhama et al. 2023) to fully automate the process and extend the tool with new mutation operators. In particular, we extend previous work by asking an LLM to generate an example program, rather than providing our own, providing automated feedback to `AFL++` for mutation-selection by constructing independent mutators, improving argument mutator to produce valid values, and introducing a type mutator. Furthermore, we test our approach with five additional local LLMs.

10 Conclusions

Finding bugs in complex simulation systems with millions of lines of code like gem5 (Binkert et al. 2011) requires new combinations of search-based strategies (such as fuzzing) and LLMs (e.g. ChatGPT) to provide extensive test cases by re-purposing and improving existing benchmarks of test programs. Although the initial complexity of preparing the simulation system for feedback-based fuzzing tools (e.g. AFL++ Michal 2023) can be discouraging, SearchSYS allows software engineers to automatically discover new errors. SearchSYS can do better than conventional fuzz testing, as the bugs it finds need not be catastrophic faults, such as segmentation errors (which fuzzers typically require), but can be a simple but automatically recognised difference in output, which is easily detected by an internal oracle (see Section 6) Langdon et al. 2017. In tandem with differential testing, we showed that it allows the discovery of masked errors.

We have proposed an improved automated approach for testing system simulators and implemented it in our prototype tool, SearchSYS. At a high level, our approach first generates programs using LLMs, which are then input to a fuzzer with our custom mutators. The architectural changes proposed in this paper are essential for achieving full automation in SearchSYS and addressing key limitations in fuzzing, as the saturation problem. The template prompt approach facilitates the automated generation of semantically rich test inputs, which are difficult to create manually. In contrast, semi-manual methods like GPT-3.5-turbo (SSBSE 2023) (Dakhama et al. 2023) require constant effort to produce diverse inputs, lack full automation, and risk saturation as the SUT adapts to previously used test inputs.

We used SearchSYS to test the gem5 simulator. We conducted 70 experiments, each repeated 10 times, using 6 LLMs, 5 software configurations, and 2 corpora variants. LLMs and fuzzing generated 101 442 issues leading to 21 new bugs in gem5, including 14 missimulations (which typically could not be detected by vanilla AFL++, which mainly targets catastrophic faults rather than missimulations). Furthermore, the gem5 developers requested access to the new test suite and intend fixing several of the bugs we have already reported to them, which provides practical validation of SearchSYS 's effectiveness.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s10515-025-00531-7.

Acknowledgements Authors are listed in alphabetical order. 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.

Author Contributions All authors contributed equally to this work. Details: All authors wrote the main manuscript text. K.E. and H.M. prepared the figures and tables. All authors reviewed the manuscript. Experimental data was collected by H.M., K.E. and A.D. and the Zenodo record was prepared using this data by K.E..

Data Availability Code and Data Availability: Our tool SearchSYS, the LLMs prompts, the experimental infrastructure, data, and results are freely available via: - SearchSYS https://github.com/karineek/SearchGEM5/ (2024) - Aidan, D., Even Mendoza, K., Langdon, W. B., Hector, M., & Justyna, P. (2024). Artifact of Enhancing Search-Based Testing with LLMs for Finding Bugs in System Simulators (ASE-Journal-V2). Zenodo. https://doi.org/10.5281/zenodo.13450472.

Declarations

Competing Interests Conflicts of Interest: Prof. Petke is Deputy Editor-in-Chief for the ASE journal.

References

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F.L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al.: GPT-4 technical report. arXiv:2303.08774 (2023)

Ackerman, J., Cybenko, G.: Large language models for fuzzing parsers (registered report). In: Proceedings of the 2nd International Fuzzing Workshop, FUZZING 2023, pp. 31–38. ACM, Seattle, WA, USA (2023). https://doi.org/10.1145/3605157.3605173

AFL compiler fuzzer. (n.d.). https://github.com/agroce/afl-compiler-fuzzer

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

Aschermann, C., Frassetto, T., Holz, T., Jauernig, P., Sadeghi, A.-R., Teuchert, D.: NAUTILUS: Fishing for deep bugs with grammars. In: Network and Distributed Systems Security (NDSS) Symposium, San Diego, CA, USA (2019). https://doi.org/10.14722/ndss.2019.23412

Aydin, Ö.: Google Bard generated literature review: Metaverse. J. AI **7**(1), 1–14 (2023)

Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: A survey. IEEE Trans. Softw. Eng. **41**(5), 507–525 (2015). https://doi.org/10.1109/TSE.2014.2372785

Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The gem5 simulator. SIGARCH Comput. Archit. News. **39**(2), 1–7 (2011). https://doi.org/10.1145/2024716.2024718

Biswas, S.: Role of ChatGPT in computer programming. Mesopotamian J. Comput. Sci. **2023**, 9–16 (2023). https://doi.org/10.58496/MJCSC/2023/002

Bossuet, L., Grosso, V., Lara-Nino, C.A.: Emulating side channel attacks on gem5: Lessons learned. In: 2023 IEEE European Symposium on Security and Privacy Workshops (EuroS &PW), Delft, Netherlands, pp. 287–295 (2023). https://doi.org/10.1109/EuroSPW59978.2023.00036

Brown, T.B., et al.: Language models are few-shot learners. (2020). https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

Bruns, N., Herdt, V., Große, D., Drechsler, R.: Toward RISC-V CSR compliance testing. IEEE Embed. Syst. Lett. **13**(4), 202–205 (2021). https://doi.org/10.1109/LES.2021.3077368

Chen, J., Patra, J., Pradel, M., Xiong, Y., Zhang, H., Hao, D., Zhang, L.: A survey of compiler testing. ACM Comput. Surv. **53**(1) (2020). https://doi.org/10.1145/3363562

Code Booga 34B. (n.d.). https://huggingface.co/oobabooga/CodeBooga-34B-v0.1

Dakhama, A., Even-Mendoza, K., Langdon, W.B., Menéndez, H.D., Petke, J.: Artifact of enhancing search-based testing with LLM for finding bugs in system simulators. Zenodo, Switzerland (2024). https://doi.org/10.5281/zenodo.13450472

Dakhama, A., Even-Mendoza, K., Langdon, W.B., Menéndez, H.D., Petke, J.: Searchgem5: Towards reliable gem5 with search based software testing and large language models. In: SSBSE 2023, Proceedings. LNCS, vol. 14415, pp. 160–166. Springer, San Francisco, CA, USA (2023). https://doi.org/10.1007/978-3-031-48796-5_14 . Best challenge track paper

Dakhel, A.M., Nikanjam, A., Majdinasab, V., Khomh, F., Desmarais, M.C.: Effective test generation using pre-trained large language models and mutation testing. arXiv:2308.16557 (2023)

Dale, R.: GPT-3: What's it good for? Nat. Lang. Eng. **27**(1), 113–118 (2021). https://doi.org/10.1017/S1351324920000601

DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: Help for the practical programmer. IEEE Comput. **11**, 31–41 (1978). https://doi.org/10.1109/C-M.1978.218136

Destefanis, G., Bartolucci, S., Ortu, M.: A preliminary analysis on the code generation capabilities of GPT-3.5 and Bard AI models for java functions. arXiv:2305.09402 (2023)

Eisele, M., Maugeri, M., Shriwas, R., Huth, C., Bella, G.: Embedded fuzzing: A review of challenges, tools, and solutions. Cybersecurity **5**(1), 18 (2022). https://doi.org/10.1186/s42400-022-00123-y

Even-Mendoza, K., Sharma, A., Donaldson, A.F., Cadar, C.: GrayC: Greybox fuzzing of compilers and analysers for C. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2023, pp. 1219–1231. ACM, Seattle, WA, USA (2023). https://doi.org/10.1145/3597926.3598130

Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., Zhang, J.M.: Large language models for software engineering: Survey and open problems. arXiv:2310.03533 (2023)

Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: AFL++: Combining incremental steps of fuzzing research. In: USENIX Workshop at WOOT 20, p. 12. USENIX Association, online (2020). https://www.usenix.org/conference/woot20/presentation/fioraldi

Fioraldi, A., Mantovani, A., Maier, D., Balzarotti, D.: Dissecting american fuzzy lop: A fuzzbench evaluation. ACM Trans. Softw. Eng. Methodol. **32**(2) (2023). https://doi.org/10.1145/3580596

gfauto. (n.d.). https://github.com/google/graphicsfuzz.git

Grattafiori, W.X., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., Synnaeve, G.: Code Llama: Open foundation models for code. arXiv:2308.12950 (2023)

Groce, A., Tonder, R., Kalburgi, G.T., Le Goues, C.: Making no-fuss compiler fuzzing effective. In: CC '22: 31st ACM SIGPLAN International Conference on Compiler Construction, pp. 194–204. ACM, Seoul, South Korea (2022). https://doi.org/10.1145/3497776.3517765

Gu, A., Rozière, B., Leather, H., Solar-Lezama, A., Synnaeve, G., Wang, S.I.: CRUXEval: A benchmark for code reasoning, understanding and execution. arXiv:2401.03065 (2024)

Herrera, A., Gunadi, H., Magrath, S., Norrish, M., Payer, M., Hosking, A.L.: Seed selection for successful fuzzing. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2021, pp. 230–243. ACM, Virtual, Denmark (2021). https://doi.org/10.1145/3460319.3464795

Hu, J., Zhang, Q., Yin, H.: Augmenting greybox fuzzing with generative AI. arXiv:2306.06782 (2023)

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 (2018). https://www.iso.org/standard/74528.html

Jain, S.M.: Hugging face. In: Introduction to Transformers for NLP: With the Hugging Face Library and Models to Solve Problems, pp. 51–67. Apress, Berkeley, CA, USA (2022). https://doi.org/10.1007/978-1-4842-8844-3_4

Javaheripi, M., Bubeck, S.: Phi-2: The surprising power of small language models. Microsoft Research Blog. (2023). https://www.microsoft.com/en-us/research/blog/phi-2-the-surprising-power-of-small-language-models/

Jiang, M., Xu, T., Zhou, Y., Hu, Y., Zhong, M., Wu, L., Luo, X., Ren, K.: EXAMINER: Automatically locating inconsistent instructions between real devices and CPU emulators for ARM. In: ASPLOS 2022: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, pp. 846–858 (2022). https://doi.org/10.1145/3503222.3507736

Kersten, R., Luckow, K., Păsăreanu, C.S.: poster: AFL-based fuzzing for java with Kelinci. In: SIGSAC. CCS '17, pp. 2511–2513. ACM, Dallas, Texas, USA (2017). https://doi.org/10.1145/3133956.3138820

Klees, G.T., Ruef, A., Cooper, B., Wei, S., Hicks, M.: Evaluating fuzz testing. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) Proceedings of the ACM Conference on Computer and Communications Security (CCS), pp. 2123–2138. ACM, Toronto, Canada (2018). https://doi.org/10.1145/3243734.3243804. Winner of the 7th NSA Best Scientific Cybersecurity Paper competition

Langdon, W.B., Yoo, S., Harman, M.: Inferring automatic test oracles. In: IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST), Buenos Aires, Argentina, pp. 5–6 (2017). https://doi.org/10.1109/SBST.2017.1

Lemieux, C., Inala, J.P., Lahiri, S.K., Sen, S.: Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In: 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Victoria, Australia, pp. 919–931 (2023). https://doi.org/10.1109/ICSE48619.2023.00085

Li, R., Allal, L.B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., et al.: StarCoder: May the source be with you! arXiv:2305.06161 (2023)

Lozhkov, A., Li, R., Allal, L.B., Cassano, F., Lamy-Poirier, J., Tazi, N., Tang, A., Pykhtar, D., Liu, J., Wei, Y., et al.: StarCoder 2 and The Stack v2: The Next Generation. arXiv:2402.19173 (2024)

Martignoni, L., Paleari, R., Reina, A., Roglia, G.F., Bruschi, D.: A methodology for testing CPU emulators. ACM Trans. Softw. Eng. Methodol. **22**(4), 29–12926 (2013). https://doi.org/10.1145/2522920.2522922

Matricardi, F.: Powerhouse in your pocket: How tiny LLMs are redefining the AI landscape. Medium. (2023). https://medium.com/@fabio.matricardi/powerhouse-in-your-pocket-how-tiny-llms-are-redefining-the-ai-landscape-fdf17718bc79

McKeeman, W.M.: Differential testing for software. Digit. Tech. J. **10**(1), 100–107 (1998)

Michal, Z.: Technical “whitepaper” for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt (Retrieved April 21, 2023)

Pahune, S., Chandrasekharan, M.: Several categories of large language models (LLMs): A short survey. Int. J. Res. Appl. Sci. Eng. Technol. **11**(7), 615–633 (2023). https://doi.org/10.22214/ijraset.2023.54677. arXiv:2307.10188

Perković, G., Drobnjak, A., Botički, I.: Hallucinations in LLMs: Understanding and addressing challenges. In: 2024 47th MIPRO ICT and Electronics Convention (MIPRO), pp. 2084–2088 (2024). https://doi.org/10.1109/MIPRO60963.2024.10569238

Petke, J.: Constraints: The future of combinatorial interaction testing. In: 8th IEEE/ACM International Workshop on Search-Based Software Testing, SBST 2015, Florence, Italy, pp. 17–18 (2015). https://doi.org/10.1109/SBST.2015.11

Pham, V.-T., Böhme, M., Roychoudhury, A.: AFLNET: A greybox fuzzer for network protocols. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), Porto, Portugal, pp. 460–465 (2020). https://doi.org/10.1109/ICST46399.2020.00062

Rajeev, R., Song, X.: An empirical study of fuzz stimuli generation for asynchronous fifo and memory coherency verification. J. Electr. Electron. Eng. **2**(3), 302–306 (2023). https://doi.org/10.33140/JEEE.02.03.13

Serebryany, K., Lifantsev, M., Shtoyk, K., Kwan, D., Hochschild, P.: SiliFuzz: Fuzzing CPUs by proxy. arXiv:2110.11519 (2021)

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al.: Llama 2: Open foundation and fine-tuned chat models. arXiv:2307.09288 (2023)

Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., Wang, Q.: Software testing with large language model: Survey, landscape, and vision. arXiv:2307.07221 (2023)

Wei, Y., Wang, Z., Liu, J., Ding, Y., Zhang, L.: Magicoder: Source code is all you need. v1 arXiv:2312.02120 (2023)

Wilk, J.: AFL's fork for fuzzing Python. (n.d.). https://github.com/jwilk/python-afl

Xia, C.S., Paltenghi, M., Le Tian, J., Pradel, M., Zhang, L.: Fuzz4All: Universal fuzzing with large language models. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24, Lisbon (2024). https://doi.org/10.1145/3597503.3639121

Yu, T., Qu, X., Cohen, M.B.: VDTest: An automated framework to support testing for virtual devices. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, pp. 583–594. ACM, Austin, TX, USA (2016). https://doi.org/10.1145/2884781.2884866

Zhang, Q., Fang, C., Xie, Y., Zhang, Y., Yang, Y., Sun, W., Yu, S., Chen, Z.: A survey on large language models for software engineering. arXiv:2312.15223 (2023)

Zhang, P., Zeng, G., Wang, T., Lu, W.: TinyLlama: An open-source small language model. arXiv:2401.02385 (2024)

Authors and Affiliations

Aidan Dakhama[1] · Karine Even-Mendoza[1] · W. B Langdon[2] · Héctor D. Menéndez[1] · Justyna Petke[2]

✉ Karine Even-Mendoza
karine.even_mendoza@kcl.ac.uk

Aidan Dakhama
aidan.dakhama@kcl.ac.uk

W. B Langdon
w.langdon@ucl.ac.uk

Héctor D. Menéndez
hector.menendez@kcl.ac.uk

Justyna Petke
j.petke@ucl.ac.uk

1 King's College London, London, England, UK

2 University College London, London, England, UK