SearchGEM5: Towards Reliable gem5 with Search Based Software Testing and Large Language Models

Aidan Dakhama[1][0009−0002−7318−7964], Karine Even-Mendoza[1][0000−0002−3099−1189], W.B. Langdon[2][0000−0002−6388−4160], Hector Menendez[1][0000−0002−6314−3725], and Justyna Petke[2][0000−0002−7833−6044]

King's[1]College London {aidan.dakhama,karine.even_mendoza,hector.menendez}@kcl.ac.uk
[2] University College London {w.langdon,j.petke}@ucl.ac.uk

Abstract. We introduce a novel automated testing technique that combines LLM and search-based fuzzing. We use ChatGPT to parameterise C programs. We compile the resultant code snippets, and feed compilable ones to `SearchGEM5`, our extension to `AFL++` fuzzer with customised new mutation operators. We run thus created 4005 binaries through our system under test, `gem5`, increasing its existing test coverage by more than 1000 lines. We discover 244 instances where `gem5` simulation of the binary differs from the binary's expected behaviour.

Keywords: AI · LLM · SBSE · SBFT · genetic improvement of tests

1 Introduction

Testing plays a key role in software's lifecycle. Today test generation is often expensive, tedious and labor-intensive. Instead, we propose to automate software testing by combining large language models (LLMs) and search-based techniques and demonstrate this on `gem5`.

Creating a test suite for `gem5` is challenging. `gem5` simulates software execution on different architectures, either processor micro-architectures or system-level. `gem5` is a large piece of code (1.34 million lines of code, LOC). Considering that the set of test inputs for `gem5` 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.

We propose a novel way of testing `gem5`. First, starting from a mix-collection of (1) industry-standard C compiler test suites and (2) tutorial programs, ChatGPT (GPT 3.5) [14] automatically creates a set of parameterized C programs with information on how to execute it. The *information* is a valid value of arguments and their data types, upon which the parameterized C program terminates normally. *Arguments* are the real values passed to the program. By compiling the LLM-generated parameterized C programs, we construct a corpus of test inputs. A *test input* is a **binary** and its **information**. Second, we extend `AFL++`, a coverage-based fuzzer [9,18]. To generate new test inputs, we have introduced custom mutators. These can modify (i) a binary using bit-flips. (ii) its arguments with their data types (e.g., mutating `0:INT32` to `55:INT32`).

To check whether our generated test inputs lead to errors in `gem5`, we use feedback from `AFL++` (crash testing) and the resultant binary run on the given architecture as an *automatic test oracle*: if `gem5` produces the same result, we deem the test run as successful. We treat it as a potential bug otherwise. We generated 4005 unique test inputs, 244 of which caused `gem5` to produce behaviour different to the one observed when the binary was run outside of `gem5`.

2 SearchGEM5

A single test input for `gem5` is composed of a binary file (--binary) and its arguments (--options). Consequently, to test `gem5`, we need a set of binaries. These binaries can be generated in two ways: 1) Compiling programs, which yield the desired binaries; 2) Alternatively, we can create binaries by mutating an existing binary. We first select a set of example programs and use a Large Language Model (LLM) to create variants that extract internal parameters as program arguments. We compile them. Subsequently, we diversify them using `AFL++` and our custom mutators.

1. **Test Input** We use LLM to generate a set of C programs suitable for testing `gem5`. We use a `BASH` script to amend minor errors.
2. **Coverage-Guided Mutation-Based Fuzz Testing** using `AFL++` [9,18].
3. **Differential Testing** Per test case, we compare the result from `gem5` to the actual test execution.

Creating a Corpus of Programs We create a corpus of parameterized test inputs. To execute a single test in `gem5`, we need: the program binary that `gem5` simulates; its arguments; and their types (e.g. 32-bit int). We generate the binary by compiling programs obtained from LLM. We also prompt LLM for a file containing the program's arguments and their types.

Training LLM to Generate Test Inputs We have three prompts to train it: 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; 3) a prompt that gives an example of a wrong response with a short explanation of what is not valid. Then we automatically construct a prompt with many programs from a single source (e.g., a single git repository):

```
    "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,...)".
```

The LLM returns pairs of programs, consisting of the original C program and its parameterized 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 `SearchGEM5` when it mutates an input.)

The sources of programs used with LLM are: c-testsuite, the LLVM test suite and C Examples. The programs that have no arguments or fail to compile are invalid. We try up to three times to automatically fix them, either using a `BASH` script for known problems (e.g., missing includes) or by asking the LLM again.

Fuzzing We use AFL++ fork [9] of the American Fuzzy Lop (AFL) fuzzer [18]. AFL++ operates by taking an initial set of files each of which is an input to the System Under Test (SUT) and instrumenting the SUT to measure test coverage. Whilst running, AFL++ uses code coverage to guide its search towards previously untested code. The next section describes our extension of AFL++ for gem5, with its complex test inputs, using a coverage-guided mutational approach.

Custom AFL++ Mutation Operators We have extended AFL++ by reimplementing the mutation operators and part of the mutation strategy though we still make use of AFL++ coverage selection criteria. This enables the selection of specific test input from the corpus and mutation of its binary file, arguments or their types, based on the test coverage data collected by AFL++ for each test input.

We have introduced two mutation operators of a test input for gem5: 1) bit-flip operator to edit a program's compiled binary file and 2) an operator to edit the value of its arguments. We do not mutate gem5 itself. Operator (2) uses type information so that the arguments remain valid.

We have implemented the extension to AFL++ in a new tool, SearchGEM5. SearchGEM5 evaluates new test inputs in the form of `binary name, arguments list, types`. SearchGEM5 then uses this information to carry our mutation operators either directly on the compiled binaries or to their arguments.

Experimental Setup The corpus of C programs was created using ChatGPT (August 3, 2023) GPT-3.5-turbo. The compiler was GCC-11, except for coverage measurements done with gcov-9 due to gem5's requirements. The Python script chosen is an example file provided by the SSBSE Challenge Track 2023 organisers, in particular, `hello-custom-binary.py`. SearchGEM5 uses AFL++ as the search engine, using its default parameters.

We ran our experiments on a single virtual machine with 8 virtual CPU Cores and 72 GB RAM, running `Ubuntu 20.04.2 LTS x86_64`. The host had a single `AMD EPYC 7313P CPU` (single socket, 3.0 GHz, 16 cores, 2 threads per CPU).

3 Results

We evaluated SearchGEM5 on its test generation capabilities, coverage, bug finding and efforts required to reproduce the results. We provide further details about the discovered bugs at [2].

Test Generation To assess the reliability of LLMs as a source of test cases for gem5 when given the prescribed test framework, we aim to evaluate: **(RQ1)** to what extent can GPT-3.5-turbo effectively generate parameterized C programs for gem5 that adhere to the specified requirements (see Section 2).

Beyond the basic prompt, we presented GPT-3.5-turbo extra examples: 1) pairs of a valid C program and a valid input for it and 2) pairs of a valid C program but with an invalid input. We ran GPT-3.5-turbo for 25 hours, often waiting due to usage limits. With GPT-4 and no subscription limits, results could be obtained within minutes. We generated **1869 C parameterized files**: 1086 compiled ok with `GCC-11 -O3`, forming a valid set of LLM-generated test programs for AFL++. AFL++ used 744 out of 1086 GPT-3.5-turbo test inputs whilst searching for new tests (AFL++ ignored 342 test inputs, usually because of invalid arguments). During 10 days run, AFL++ generated further **2136** unique mutated test inputs. *In total*, SearchGEM5 generated **4005** unique tests, with 3222 of them becoming binaries.

Test Coverage Our objective is to evaluate the gap between LLM-generated test inputs and the overall coverage achieved by our hybrid approach, which combines LLMs with AFL++ search for additional coverage leveraging our novel mutation operators. **(RQ2)** What is the coverage of LLM- and AFL++-generated test inputs? Can AFL++ improve the basic coverage achieved by LLM-generated ones in gem5?

For coverage measurements, we used 1086 and 2136 distinct binaries generated using LLM and AFL++, respectively. We built gem5 with g++ 9.4 -O1 and gcov, adding gcov instrumentation overheads. We measured a smaller part of the gem5 codebase, i.e. that relevant only to X86. We used the gcov-based tool gfauto[10] to generate the coverage results in a human-readable format for 3380 files in the gem5 codebase (including header and system header files). The LLM-generated test inputs achieved a total of 39,143 lines of coverage on the gem5 codebase. While our AFL++ covered 40,337 lines, an extra 1,194 lines (inclusive).

Bug Finding in gem5 Our primary objective here is to estimate the effectiveness of our approach in uncovering bugs within gem5. That is, **(RQ3)** how effective is our approach at finding bugs in gem5?

We have found panic crashes, assertion violations, crashes, hangs and mis-simulation bugs in gem5 summarized in Table 1. We have investigated and classified all the crashes, identifying two different types of assertion violations and 10 different panic crashes in the gem5 codebase. Although the hangs may be caused by several different issues, they are grouped together in Table 1 (none of them are associated with pending inputs or lack of resources).

For example, a test input on line #14 of Table 1 was generated by GPT-3.5-turbo. It was incorrectly simulated by gem5 (running without simulating the operating system as well as the program, SE mode), resulting in a different output than the native X86 run. During the simulation with gem5, with an invalid input, the program terminated wrongly with an exit code 0. However the same combination of program and bad input led to the program, on reaching line 11 when run on a native X86 being terminated with a segmentation fault (LLM-generated program executed as `./00172.c.o 0`; available at [2]). It appears that the address space of gem5 might mask these kinds of pointer errors, similarly to what happens with virtualization for obfuscations [15], where invalid addresses that normally belong to the operating system are part of the process address space.

Portability AFL++ has been applied to various targets [1,12,16,17] and hence a different target SUT should be easy. Our custom AFL++ mutator can be re-used since we are doing target-independent bit-level mutations.

GPT-3.5-turbo has been trained on a wide range of code from many programming languages, such as Python and Java. By adjusting the LLM prompt and program examples to use the desired language, our method can easily be adapted to the generation of test inputs in other programming languages [5,8].

Table 1: List of errors in gem5 found with SearchGEM5-generated test inputs. Mutation operations: **B**=Binary bit-flip; **C**=Constants bit-flip of arguments; **B/C**=B and/or C; **B (C optional)**=B with or without C; **ANY**=LLM-generated or B/C. #**Input**: distinct test inputs of the same bug.

#	Error Kind	Operations	#inputs	Details
1	Panic error	B	1	File exec-ns.cc.inc, line 17, "attempt to execute unimplemented instruction 'femm' (...)".
2	Panic error	B/C	52	File sim/faults.cc, line 60, "panic condition !FullSystem occurred: fault (General-Protection) detected (...)".
3	Panic error	B	4	File base/loader/elf_object.cc, line 129, "gelf_getphdr failed for segment 0 (...)".
4	Panic error	B	3	File base/loader/memory_image.hh, line 70, "panic condition offset + size >ifd->len() occurred (...)".
5	Panic error	B	1	File cpu/simple/timing.cc, line 953, "panic condition pkt->isError() occurred: Data access (...)".
6	Panic error	B (C optional)	26	File arch/x86/faults.cc, line 131, "Unrecognized/invalid instruction executed (...)".
7	Panic error	B	1	File arch/x86/faults.cc, line 164, "Tried to execute unmapped address (...)".
8	Panic error	B (C optional)	14	File arch/x86/faults.cc, line 166, "Tried to execute unmapped address (...)".
9	Panic error	B/C	61	File arch/x86/faults.cc, line 166, "Tried to read unmapped address (...)".
10	Panic error	B/C	12	File arch/x86/faults.cc, line 166, "Tried to write unmapped address (...)".
11	Crash (assert fail)	B	2	File base/loader/elf_object.cc, line 80, "virtual gem5::loader::ObjectFile* gem5::loader::ElfObjectFormat (...)".
12	Crash (assert fail)	B	1	File base/loader/elf_object.cc, line 311, "void gem5::loader::ElfObject::determineOpSys(): Assertion (...)".
13	Hangs	B/C	6	gem5hangs with a timeout of 500 s on small programs.
14	Mis-simulation	ANY	56	Invalid program is simulated as valid a program.
15	Mis-simulation	B	1	Variable's value is random in X86 but fixed in simulation.
16	Crash	C	1	X86 terminates normally but the simulation failed to parse input arguments (UnicodeDecodeError).
17	Mis-simulation	C	2	X86 and simulation had different outputs.

4 Conclusions

Finding bugs in complex simulation systems like gem5 [4] requires new combinations of search-based strategies (like fuzzing) and LLMs like ChatGPT to provide extensive test cases by re-purposing and improving existing benchmarks of test programs. Although the initial complexity can be discouraging because preparing the simulation system for feedback-based fuzzing tools, like AFL++ [18], requires the instrumentation of the whole system, it allows us to automatically discover new errors, which need not be the catastrophic faults, such as segmentation errors, which fuzz testing usually demands, but can be a simple but automatically recognized difference in output, which is easily detected by an internal oracle (see Section 3) [13]. In tandem with differential testing, we showed that it allows masked errors discovery. Our approach uses AFL++ in an unconventional way, replacing test input fuzzing with a more sophisticated input format as discussed in Section 2, an idea which was adopted by various domains such as compiler testing [1,3] or network protocol analysis to manipulate network protocols for fuzzing [16]. In terms of testing `gem5`, while there are efforts in verifying its architectural compliance [7] and the integrity of its code, `gem5` testing or verification approaches [6] that can go deeper into its internals, such as `SearchGEM5`, are essential to validate its many possible options. The portability discussion and Table 1, which quantifies instances of test inputs per bug, offer partial insights on bug reproducibility limitations stemming from non-determinism in random testing and LLMs (Section 3). We defer further analysis to future research.

Data Availability `SearchGEM5`, the LLMs prompt and the experimental infrastructure, data, and results are freely available via [2,11].

Acknowledgments Authors are listed in alphabetical order. This work was supported by the UKRI EPSRC grant no. EP/P023991/1 and the UKRI TAS Hub grant no. EP/V00784X/1 and EP/V026801/2.

References

1. AFL compiler fuzzer: `https://github.com/agroce/afl-compiler-fuzzer`
2. Artifact of SearchGEM5. Zenodo (2023). https://doi.org/10.5281/zenodo.8316685
3. Aschermann, C., Frassetto, T., Holz, T., Jauernig, P., Sadeghi, A.R., Teuchert, D.: Nautilus: Fishing for deep bugs with grammars. In: NDSS (2019)
4. Binkert, N., et al.: The gem5 simulator. ACM SIGARCH computer architecture news **39**(2), 1–7 (2011)
5. Biswas, S.: Role of ChatGPT in computer programming.: ChatGPT in computer programming. Mesopotamian Journal of Computer Science **2023**, 8–16 (Feb 2023)
6. Bossuet, L., Grosso, V., Lara-Nino, C.A.: Emulating side channel attacks on gem5: lessons learned. In: EuroS&PW. pp. 287–295. IEEE (2023)
7. Bruns, N., Herdt, V., Große, D., Drechsler, R.: Toward RISC-V CSR compliance testing. IEEE Embedded Systems Letters **13**(4), 202–205 (2021)
8. 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 (2023)
9. Fioraldi, A., et al.: AFL++ : Combining incremental steps of fuzzing research. In: USENIX Workshop at WOOT 20. USENIX Association (2020)
10. Git repository of gfauto,: `https://github.com/google/graphicsfuzz.git`
11. Git repository of searchGEM5: `https://github.com/karineek/SearchGEM5/`
12. Kersten, R., Luckow, K., Păsăreanu, C.S.: POSTER: AFL-based fuzzing for java with kelinci. In: SIGSAC. p. 2511–2513. CCS '17, ACM (2017)
13. Langdon, W.B., Yoo, S., Harman, M.: Inferring automatic test oracles. In: SBST. pp. 5–6. Buenos Aires, Argentina (22-23 May 2017)
14. Lund, B.D., Wang, T.: Chatting about ChatGPT: how may AI and GPT impact academia and libraries? Library Hi Tech News **40**(3), 26–29 (2023)
15. Menéndez, H.D., Suárez-Tangil, G.: ObfSec: Measuring the security of obfuscations from a testing perspective. Expert Systems with Applications **210**, 118298 (2022)
16. Pham, V.T., Böhme, M., Roychoudhury, A.: AFLNET: A greybox fuzzer for network protocols. In: ICST. pp. 460–465 (2020)
17. AFL's' fork for fuzzing pure Python: `https://github.com/jwilk/python-afl`
18. Zalewski M.: Technical "whitepaper" for afl-fuzz. `http://lcamtuf.coredump.cx/afl/technical_details.txt` (Retrieved April 21, 2023)