Fuzzing-Based Differential Testing For Quantum Simulators

Daniel Blackwell[1][0000−0001−7320−9057], Justyna Petke[1][0000−0002−7833−6044], Yazhuo Cao[2][0009−0002−1201−9908], and Avner Bensoussan[2][0009−0007−3285−9468]

[1] University College London, UK {daniel.blackwell.14 | j.petke}@ucl.ac.uk
[2] King's College London, UK {yazhuo.cao | avner.bensoussan}@kcl.ac.uk

Abstract. Quantum programs are hard to develop and test due to their probabilistic nature and the restricted availability of quantum computers. Quantum simulators have thus been introduced to help software developers. There are, however, no formal proofs that these simulators behave in exactly the way that real quantum hardware does, which could lead to errors in their implementation. Here we propose to use a search-based technique, grammar-based fuzzing, to generate syntactically valid quantum programs, and use differential testing to search for inconsistent behaviour between selected quantum simulators. We tested our approach on three simulators: Braket, Quantastica, and Qiskit. Overall, we generated and ran over 400k testcases, 2,327 of which found new coverage, and 292 of which caused crashes, hangs or divergent behaviour. Our analysis revealed 4 classes of bugs, including a bug in the OpenQASM 3 `stdgates.inc` standard gates library, affecting all the simulators. All but one of the bugs reported to the developers have been already fixed by them, while the remaining bug has been acknowledged as a true bug.

Keywords: Differential Testing · Fuzzing · Quantum Simulators

1 Introduction

Quantum computers offer an exciting opportunity to massively speed up existing computation. However, writing valid quantum programs is non-trivial. By shifting the paradigm from traditional computing the outputs are no longer deterministic. It is thus no wonder that testing quantum software is a challenging task [1]. Furthermore, quantum computers are not widely available, require specialist knowledge to run and maintain, and suffer with inaccuracy due to noise.

Quantum simulators have been introduced to ease developers in programming and validating quantum circuits. Nevertheless, there are no formal guarantees that the outputs of simulation will be the same on real quantum hardware. Furthermore, simulators themselves might not be bug-free. In fact, Wang et al. [5] generated semantics-preserving gate transformations and showed that in 33 of 730 cases the outputs of simulations diverged. The technique, however, requires specialist domain knowledge to create these metamorphic relations; we lack the knowledge to ascertain whether they have already found all of the viable ones.

To aid the development of quantum simulators we propose to use search-based differential testing to check their validity. In particular, we investigate the difference in behaviour between different quantum simulators, when given the same quantum programs, generated by a grammar-based fuzzer. As far as we know, **this is the first time our differential search-based testing approach has been applied to test quantum program simulators**.

Our initial results are encouraging. We found 4 classes of bugs, with at least one in all three quantum program simulation frameworks tested, i.e., Qiskit[3], Braket[4] and Quantastica[5]. **All bugs have been confirmed and all but one already fixed** by the developers. Furthermore, our method has generated a large number of testcases (over 400k) covering a lot of functionality which we have minimised to a set of 842 testcases that achieve 100% of the coverage discovered by the fuzzer; this may be useful to the simulator maintainers as a standalone regression test set, or alternatively our approach could be trivially adapted to do regression fuzz testing. Moreover, our methodology can aid in extending existing quantum program benchmarks. To allow further uptake of our approach we provide the artefact at `https://doi.org/10.5281/zenodo.11002154` and the GitHub repo: `https://github.com/GloC99/fuzzingQuantumSimulator`.

2 Differential Fuzz Testing

Here we propose to perform differential fuzz testing of quantum simulators. We first take existing quantum programs representing valid circuits. We compile them into an intermediate representation: the OpenQASM 3 language [2] (herein referred to as QASM). We feed these programs to a search-based automated test generation tool to generate more testcases. We chose to use an existing grey-box fuzzer for this purpose. Grey-box fuzzing is a search-based software testing technique that generates new testcases by mutating existing ones; coverage feedback is used to retain any new testcases that exercise new program functionality. In essence, the mechanism of a grey-box fuzzer can be likened to a genetic algorithm whose fitness function is the total coverage of the retained set of testcases. We have extended the fuzzer with a QASM grammar-aware mutator. This way we generate only syntactically valid programs. Finally, we feed the generated QASM programs through different quantum simulators, observe crashes or hangs where they occur, and compare the outputs where they don't.

In order to evaluate our approach we built tooling based on the AFL++ fuzzer [3] and tested it on 3 quantum simulation frameworks: Qiskit, Braket and Quantastica. Next, we detail each step of the implementation of our approach.

Grammar Mutator We use `Grammar-Mutator`[6] from AFL++'s set of included custom mutators. As it takes grammars provided in an unusual JSON format; we

[3] `https://github.com/Qiskit/qiskit-aer`
[4] `https://github.com/amazon-braket/amazon-braket-default-simulator-python`
[5] `https://github.com/quantastica/quantum-circuit`
[6] `https://github.com/AFLplusplus/Grammar-Mutator`

had to manually adapt the ANTLR4 grammar from the OpenQASM 3 specification[7]. Additionally, we found that the specification is very much forward-looking, and 10 of the 28 statements are not yet implemented in the two simulators supporting OpenQASM 3. As such, these statements along with certain other unimplemented features were removed from our grammar in order to increase the number of valid executable programs we could generate. `Grammar-Mutator` works on a context-free grammar, which means that while it generates programs that will pass the lexing stage, many of these will not successfully parse due to invalid semantics. To increase the likelihood of successful parsing, and thus execution, we adjusted the grammar so that the generated program is guaranteed to begin with the declaration of a quantum register, and will include at least one gate statement; without these, the Braket simulation framework we use would throw an exception to indicate that the program has no functionality.

Instrumentation We chose to use `python-afl`[8] in combination with AFL++, in order to have easy access to a grammar mutator. As noted in GitHub issue no. 25 by the author, the instrumentation method used by `python-afl` incurs significant runtime overhead. Running the three simulators with example programs took around 0.3-0.5 seconds already; with instrumentation this could be up to 2 or more seconds. To mitigate this, we added functionality to `python-afl` to allow us to enable and disable instrumentation at points in the fuzzing harness so as to eliminate the overhead of instrumentation on 'boring' functionality. This allowed us to average approximately 1.1 executions per second – still slow by fuzzing standards, but several times better than the naive approach.

It is worth noting that the Quantastica `quantum-circuit` simulator is written in JavaScript and hence cannot be invoked directly from the Python fuzzing harness. We built a Node JS server that is run on localhost, with a single endpoint to receive a QASM program as a string, load it in, execute it and send the result as a response. The fuzzing harness sends a web request and parses the response in order to compare with that of the other simulators; as a result, no coverage feedback is available for this simulator.

Fuzzing Harness A single fuzzing harness was created that took the OpenQASM 3 input generated by the fuzzer, and fed it through all three simulators in the following order: Braket, Qiskit then Quantastica; collecting the state vector for each. As Quantastica only supports OpenQASM 2 (rather than the OpenQASM 3 that our generated programs are in), we chose to export the circuit that we have just generated in Qiskit to OpenQASM 2; this is not an ideal solution as any parsing errors made by Qiskit will be passed along, but it is the best option we have short of not testing Quantastica. Chaining the executions in the way we do means that if a crash occurs in Braket, then the input will not be ran for Qiskit or Quantastica; when triaging the discovered crashes we run the generated programs on the other simulators to ensure that the same crash does not occur for them.

[7] `https://openqasm.com/versions/3.0/grammar/index.html`
[8] `https://github.com/jwilk/python-afl`

Any generated programs that parse and execute successfully on all simulators make it to the final step, which is comparison of the output probability distributions. For this we used Jensen-Shannon divergence [4], which is a finite symmetric measure of divergence between two probability distributions. Using an assertion, we set an arbitrary cap of 0.01 to be allowed when comparing Braket with Qiskit, and Qiskit with Quantastica; failing to pass this assertion marks this input as a crash in the fuzzer. We could set such a strict cap on expected divergence because directly using the state vectors avoids noise and should be fully deterministic. We did not cap at 0 in order to allow for a small amount of accumulated floating point errors.

Initial Seeds To obtain the initial seeds we took the set of 382 benchmarks in OpenQASM 3 form from the `mqtbench`[9] benchmarks. We minimised them down to a subset of 22 testcases that covered all edge cases found in the complete set using `py-afl-cmin`[10]. We modified these testcases to minimise the number of unique variable identifiers and added these to the grammar in order to increase the probability of generating programs free from undeclared variable usage.

Fuzzing Campaigns Many short campaigns were run whilst creating and debugging the pipeline, often these produced large numbers of crashes due to unhandled exceptions during the parsing phase. We noted down each unique crash type and decided whether it was handled sufficiently gracefully or not; we added catches within our fuzzing harness for those exceptions that we believe were handled gracefully within the quantum library, as our aim was to test the functionality of the simulators rather than our tools ability to create valid QASM programs. In some cases we found exceptions during the parsing phase that were not well handled and could benefit from providing more context to the user. We report results from our longest run campaign which was performed on a 2023 Macbook Air with M2 processor and 8GB RAM; this ran for 106 hours on a single core.

3 Results

Our longest fuzzing campaign resulted in 407k executions (and thus approximately as many unique QASM programs), generating a corpus of 2,327 testcases (from an original 22), 139 saved crashes and 153 saved hangs; note that these are AFL++ statistics where only inputs that cover new functionality are saved, thus the resultant testcases significantly increase the diversity of the orignal 22 tests.

We discovered that 4 types of bugs were responsible for all 139 crashes; they are listed in order of severity in the rest of this paragraph. One bug filed to Qiskit maintainers ended up being an error in the specification for the OpenQASM 3 standard library (`stdgates.inc`); this directly led to us publicly filing a report in Braket and privately alerting Quantastica to a potential error in their simulator. Qiskit's standard library implementation has now been fixed, and Braket's simulator too. Additionally, we reported a significant performance issue in Braket, which has now been fixed; and a crash in the Quantastica simulator which has also been fixed. We describe these bugs further:

[9] `https://www.cda.cit.tum.de/mqtbench/`

[10] `https://github.com/jwilk/python-afl/blob/master/py-afl-cmin`

Braket In Braket we found that the QASM interpreter made certain assumptions about the available attributes of some objects; this resulted in uncaught exceptions that provided no context about where the error occurred. In most cases, these were invalid programs and our only concern was that the error message was less helpful than others which provided context about which line and token the interpreter failed on. It is certainly possible to rationally argue that this is a whole family of bugs (though we do not), and we have witnesses to eight different crash locations.

`<Bug BRAKET1>`: In one of these cases the interpreter did not conform to the OpenQASM 3 specification, and after reporting we are assured that this will be addressed in a future release.

`<Bug BRAKET2>`: We filed another bug report whereby simulating relatively simple quantum circuits of 14 qubits resulted in the Python interpreter being killed due to running out of memory. This was due to an error in the implementation of the `gphase` builtin instruction, and has now been fixed by a maintainer; after which the simulator could comfortably handle the same circuits with 25 or more qubits.

Qiskit For consistency between platforms, we chose to manually expand the `include "stdgates.inc"` statement using a simple string replacement with the file definition from the original OpenQASM 3 specification publication [2]. All three simulators provide inbuilt definitions for the standard gates, however, as a direct result of forcing them to have to generate the definitions directly from QASM we discovered that Qiskit's output probability distributions diverged from the other simulators for testcases involving use of the `sx` gate.

`<Bug QISKIT1>`: After triaging and discovering that the divergence only occurred when manually specifying the `sx` gate definition in QASM (rather than relying on the built-in definition), we decided to file an issue. The maintainers narrowed it down to one line: `gate sx a { pow(1/2) @ x a; }`. According to the OpenQASM 3 specification, dividing two integer literals should use integer division resulting in `1/2` resolving to `0`; whereas Braket performed float division resulting in a value of `0.5`. Ultimately it was decided that the error was in the original `sx` gate definition as provided in the specification and it should instead have either written the fraction as `1.0/2` (or equivalent) or used `0.5`. While this meant that the `stdgates.inc` (the equivalent of OpenQASM 3's standard library) would need correcting, it also meant that Braket has an implementation error in applying floating point division where integer division should be used. Quantastica's behaviour aligned with Qiskit due to the OpenQASM 2 code being generated by Qiskit re-exporting the circuit that it had produced – after realising the divergent behaviour in Braket, we manually constructed a simple testcase to check Quantastica and found that it too uses floating point division. We filed an issue with Braket to alert them to the issue, and privately informed the Quantastica maintainers of the finding, though as the OpenQASM 2 specification is less rigorous and does not directly specify integer division, we are not treating this as a bug. The Qiskit maintainers have merged a fix to the `stdgates.inc` standard library file, and Braket maintainers have fixed the floating point division error.

Quantastica `<Bug QUANT1>`: The Quantastica simulator does some undocumented processing of register identifiers, causing some of our generated testcases to throw an out-of-heap-memory error. Identifiers that trigger this bug are also generated by Qiskit when exporting to OpenQASM 2 – this is not a highly improbable bug to encounter. The online tools `quantum-circuit` and `q-convert` run this JavaScript code in the browser, and attempting to parse code containing the bad identifiers results in a hang rather than a crash. As this bug could be used in a denial-of-service attack, we first emailed the library maintainer to ensure that no server-side applications could be targeted and only filed a publicly visible issue once we had assurance that all applications were run client-side.

3.1 Effectiveness of the Differential Testing Approach

Our search-based fuzz testing approach revealed multiple crashes and hangs. We can directly attribute the differential testing approach to the discovery of `<Bug BRAKET2>` causing crashes for gphase instructions with 14 qubits, and `<Bug QISKIT1>` where floating point division was incorrectly used in place of integer division. As many of the auto-generated circuits had large numbers of qubits, many testcases crashed or were killed, so if we were just testing Braket alone we may not have realised there was an issue. It was only after we spotted that Qiskit correctly handled one particular testcase that Braket crashed on that we decided to investigate further and discovered that there was probably an implementation issue. In the case of the division bug, it was as a direct result of the Jensen-Shannon divergence bounding assert being failed that this was detected. Testing any of the simulators on their own could have only discovered this with an appropriate oracle – which we do not have.

4 Conclusions

We proposed to use search-based differential testing to check validity of quantum program simulators. In particular, we used grammar-aware fuzzing to generate valid programs, which were then fed into different quantum simulators. Our results from over 400k executions show that our approach is useful in finding real bugs in such software. **Funding** We thank the ERC Advanced Grant no. 741278 and UK EPSRC Grant no. EP/S022503/1.

References

1. de la Barrera, A.G., de Guzmán, I.G.R., Polo, M., Piattini, M.: Quantum software testing: State of the art. J. Softw. Evol. Process. **35**(4) (2023)
2. Cross, A., Javadi-Abhari, A., Alexander, T., De Beaudrap, N., Bishop, L.S., Heidel, S., Ryan, C.A., Sivarajah, P., Smolin, J., Gambetta, J.M., Johnson, B.R.: OpenQASM 3: A broader and deeper quantum assembly language. TQC **3**(3), 1–50 (2022)
3. Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: AFL++: Combining incremental steps of fuzzing research. In: WOOT Workshop. USENIX Association (2020)
4. Menéndez, M., Pardo, J., Pardo, L., Pardo, M.: The Jensen-Shannon divergence. Journal of the Franklin Institute **334**(2), 307–318 (1997)
5. Wang, J., Zhang, Q., Xu, G.H., Kim, M.: QDiff: Differential Testing of Quantum Software Stacks. In: ASE Conf. pp. 692–704. IEEE (2021)