Poster: Fuzz Testing of Quantum Program

Jiyuan Wang∗, Fucheng Ma∗, Yu Jiang∗
∗*School of Software, Tsinghua University, Beijing, China*

***Abstract*—Nowadays, quantum program is widely used and quickly developed. However, the absence of testing methodology restricts their quality. Different input format and operator from traditional program make this issue hard to resolve.**

In this paper, we present `QuanFuzz`, a search-based test input generator for quantum program. We define the quantum sensitive information to evaluate test input for quantum program and use matrix generator to generate test cases with higher coverage. Because of the impossibility of copying qubit, we record the operations which lead initial seeds to test inputs instead of recording qubits themselves. First, we extract quantum sensitive information – measurement operations on those quantum registers and the sensitive branches associated with those measurement results, from the quantum source code. Then, we use the sensitive information guided algorithm to mutate the initial input matrix and select those matrices which improve the probability weight for a value of the quantum register to trigger the sensitive branch. `QuanFuzz` on benchmarks and acquired 20% - 60% more coverage compared to traditional testing methods.

I. Introduction

Quantum computer is being commercialized and applied in various areas [1], and rapid progress has been made in quantum programming language. In particular, quantum programming languages have been increasingly developed for nearly twenty years such as QCL [2], QPL [3], $Q\,|SI\rangle$ [4], Q#[5]. For example, Q$|SI\rangle$ is a platform created in $.NET$ language to support quantum programming using a quantum extension of the while-language. The framework of the platform includes a compiler of the quantum while-language and a suite of tools for simulating quantum computation, optimizing quantum circuits, and analyzing quantum programs.

Quantum program computation logic is embedded in the quantum registers, quantum gates and measurement results of those quantum registers [6]. In quantum programs, measurement operation $measure(q)$ of the same quantum register q can produce different results in different executions. Because of this huge difference between quantum program and traditional program, traditional software validation methodologies can not be applied to quantum program directly.

Some researchers have customized traditional verification techniques to verify quantum programs. For example, QPMC [7], a model checker for quantum program, is able to take the state space in the classical way by using Quantum Markov Chain, and apply classical model checking on quantum program. Those verification techniques are accurate, but they can easily run into the state explosion problem for complex quantum programs with large number of quantum registers.

Challenges: An alternative way is testing. The common practice of fuzz testing is to measure coverage information on a test input and capture crashes[8], [9]. However, there are three challenges in fuzz testing of quantum programs. First, since the difference between the logic of quantum program and traditional program, branch or path guiding information used in traditional fuzz testing is not suitable for quantum program. In other words, it is difficult to figure out the kernel information to evaluate the test input. Second, since quantum mechanics implies that copying and assigning qubits are impossible, how to record the information and input during fuzzing is also a complex problem. Finally, since the state of quantum registers is complex and the operation gates on those registers are also with many types, it is hard to generate test input for quantum programs efficiently.

Approach: In this paper, we propose `QuanFuzz` to address the above challenges, which is the first initial exploration to do fuzz testing of quantum programs. We define the quantum-sensitive information (including quantum register measurement, sensitive branch), and propose a greybox fuzz testing model aiming to generate inputs to change the state of quantum registers and maximize the coverage for a given quantum program. In order to record information during the process, we record operations and changes instead of quantum registers themselves. `QuanFuzz` uses matrix generator to mutate the input matrices and select matrices with higher probability weights for the value of quantum registers to trigger the quantum sensitive branches. During the mutation process, `QuanFuzz` keeps several matrices with the top weights and continuously updates the matrices by traversing all qubits crossing random gates. In this way, with a few iterations, `QuanFuzz` is able to obtain rare inputs, and can automatically choose the better input to trigger the quantum-sensitive branches and detect cashes. We evaluate `QuanFuzz` on the benchmarks provided in $Q\,|SI\rangle$. Compared to the traditional testing methods, `QuanFuzz` increases branch coverage by 20%-60%, especially on those quantum sensitive branches.

II. Background and Motivation

A. Background on Quantum mechanics

Quantum systems are represented through a normalized complex Hilbert space, which is a completed vector space over field $\mathbb{C}^k$ with an inner product: $H \times H \to \mathbb{C}$.

In this paper, we can simply set $k = 2^n$ where n is the number of quantum bit (qubit). The state of a qubit is either 0 or 1. Therefore, for a quantum register contains 1 qubit, $k = 2^1 = 2$. The state then can be represented as:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, |\alpha|^2 + |\beta|^2 = 1, \alpha, \beta \in \mathbb{C} \qquad (1)$$

where ψ is the total state function, α and β are the probability of each state. $|\psi\rangle$ is called a Dirac notation, which represents a vector in Hilbert space. The vector is called a ket, while its conjugate transpose $\langle\psi|$ is called a bra.

A system with n qubits quantum register has 2^n states, and its information cannot be read directly. Only after measurement, it will be in one determinate state. Consider $|\psi\rangle$ as an example, the state will be in 0 with probability $|\alpha|^2$and in 1 with probability $|\beta|^2$. And it is obvious that we should set total probability $|\alpha|^2+|\beta|^2=1$. When a quantum register contains n qubits, the k will be 2^n and its state will be:

$$|\psi\rangle=\sum_{i=0}^{2^n-1}c_i|i\rangle \quad with \sum_{i=0}^{2^n-1}|c_i|^2=1 \tag{2}$$

Basic operators of quantum computing logic are called unitary gates, which are corresponding to the logic gates(e.g. and, or, xor) in traditional computer systems and are usually represented by matrices. The quantum programs mainly use these gates to change the value of qubits. We show commonly use gates for one qubit below [10].

$$X=\begin{pmatrix}0&1\\1&0\end{pmatrix} \quad Y=\begin{pmatrix}0&-i\\i&0\end{pmatrix} \quad Z=\begin{pmatrix}1&0\\0&-1\end{pmatrix}$$
$$H=\frac{1}{\sqrt{2}}\begin{pmatrix}1&1\\1&-1\end{pmatrix} \quad S=\begin{pmatrix}1&0\\0&i\end{pmatrix} \quad T=\begin{pmatrix}1&0\\0&e^{\frac{i\pi}{4}}\end{pmatrix}$$

The X gate, Y gate, and Z gate are called Pauli gates. X gate is the quantum equivalent of the NOT gate for classical computers, while Y gate and Z gate equates to a rotation around the Y-axis and Z-axis of the Bloch sphere by π radians. S gate and T gate work as two other rotations. Consider H gate as an example, which is called Hadamard gate[11]. It represents a rotation of π about the axis $(\hat{x}+\hat{z})/\sqrt{2}$. To get the result after H gate, we simply multiple the H on the left side of one qubit state function as below:

$$|\varphi\rangle=H|\psi\rangle \tag{3}$$

B. Motivation Example of Quantum Sensitive Coverage

We use an example programmed by QCL, the first quantum programming language invented in 1998 [2], to show the difference between quantum program and traditional program.

```
1  procedure example(){
2  //define a quantum register with 5 qubits
3          qureg q[5];
4  //make all states have the same probability
5          Mix(q);
6  //measure the value of q[5] and check
7          if (measure(q)==5){
8          //quantum sensitive branch
9                  int i=1/0;    //bug code
10         }
11 }
```

In the code, $q[5]$ is a quantum register with 5 qubits. $measure(q)$ is the measurement function. As pointed out before, the result of this function could be any value S from 0 to 31, and the probability to be the value S equals the corresponding matrix element's square. Specifically, in this program, the result of $measure(q)$ could be 0 to 31 with equal probabilities $\frac{1}{2^5}$ because of the gate operation denoted as Mix(). If and only if $measure(q)$ equals 5, the branch can be executed and the bug can be detected. But it has only $\frac{1}{2^5}$ chance to happen. Additionally, according to section II.A, we know even with same q, $measure(q)$ can give different results, which gives more difficulty for testing.

In order to better test the quantum program, we aim to use a guided matrix generator to generate the test input, not just traditional fuzzing. In this example, it means to use matrix generator to make $measure(q)==5$ more likely to happen, and the quantum sensitive branch in line 7 could be triggered.

III. Proposed Approach

The overview of of `QuanFuzz` is described in Figure 1. We firstly analyze the source code to get quantum sensitive information, including the measurement operation and the sensitive branches related to the results of measurement. Then, focusing on quantum sensitive parts, we use the matrix generator to get the test cases to satisfy the condition. With an original matrix S, the traversing algorithm applies quantum gates to mutate and get more matrices. Since the change of qubit can not be recorded, we record operation instead. Then we evaluate the matrices by their probability weights for the sensitive value of the quantum register and store good matrices in matrix queue. If one of the matrices' weights for the value of a quantum register is larger than the threshold p, then we find a good test input. If not, several candidate matrices will be regarded as new S for the next iteration.

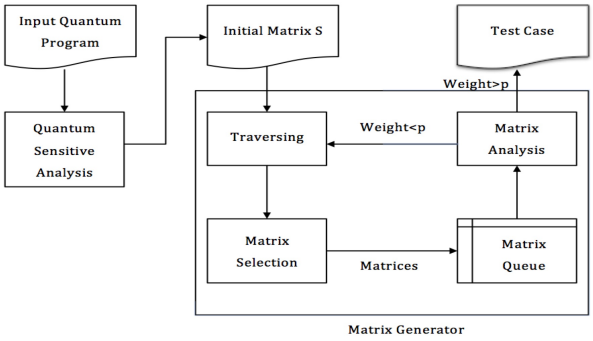

Fig. 1. The overall workflow of `QuanFuzz`, include the quantum sensitive analysis, and matrix generator based on the guided sensitive information.

A. Quantum Sensitive Analysis

As described in the motivation example, although the quantum program code structure is similar with traditional program code standard, the traditional branch or path coverage are closely related with the quantum measurement operation. We need to pay more attention to these quantum parts, like $measure$, H gate, etc, and look into the probability weight for the value of quantum register. In particular, we extract three types of quantum sensitive information from the quantum program source code: ket information, measurement information (i.e. measurement operator) and oracle information (i.e. sensitive results and operations on the measurement value).

In order to extract the quantum sensitive information, we instrument the source code at four parts – input matrix read, transform ket with input matrix, ket before measurement output, and measurement result output. We extract ket information and store in ketSet, extract measurement information and store in the corresponding ket's container. Those chunks of information are used to guide the test input generation especially for the input matrix selection and mutation.

B. Matrix Mutation and Selection

The core process of the mutation and selection is presented in Algorithm 1. We use the six most commonly used basic quantum gates presented in the section II.A as the basic transformation gate mutators.

At first, the matrix queue $Top\_Matrices$ only has one input which is exactly the initial matrix S. Next, `QuanFuzz` traverses every qubit using $traversing()$ function and obtains the new matrices with their probability weights for each value of the quantum register. Function $traversing(S, k, n)$ traverses matrix S from k^{th} qubit to n^{th} qubit. For each qubit, we randomly apply 2 gates on it. Take 2 qubits as an example, the workflow of traversing is described in Figure 2, and each qubit is sequentially operated by two selected quantum gates to generate the candidate matrices. The record process is described in Figure 3, showing how matrix queue stores operation on qubits instead of qubits themselves. After traversing all qubits of matrix S, we put the new matrices with their corresponding probability weight in $Top\_Matrices$.

Algorithm 1 main(S)

Input: S ← original matrix
p ← the probability to trigger the sensitive branch
Output: Best matrix to execute the sensitive branch

```
1:  Top_Matrices=[]  //store six best matrices and their weight
2:  Top_Matrices.append=([], Weight_Analysis([], desired, n))  //add seed
3:  while Top_Matrices[0].weight<p do
4:    for i=1 to min(Top_Matrices.totalnumber(),6) do
5:      traversing(Top_Matrices[i],1,n)  //traverse all qubits
6:    Top_Matrices.sort()  //sort by matrix weight
7:    for i=6 to Top_Matrices.totalnumber() do
8:      Top_Matrices.delete(i)  //only store six best matrices
9:    iteration_time ++
10: return Top_Matrices[0].matrix
```

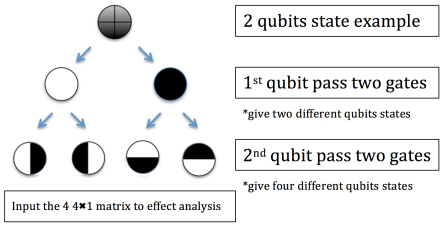

Fig. 2. 2 qubit state example, for gate transformation of each qubit

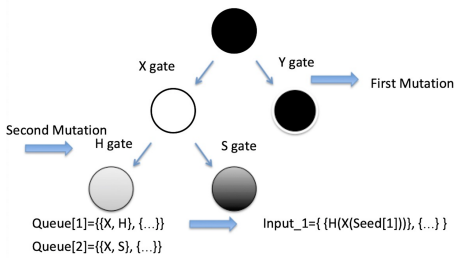

Fig. 3. 1 qubit record example, for operation record of each qubit

The reason to use the search-based algorithm and sampling traversing is that the operation on those registers is with huge space. It is impossible to go through all the possible states. Finally, if the probability weight is larger than threshold p, we stop the iteration and return the corresponding matrix. In contrast, we let $Top\_Matrices[1]$ to $Top\_Matrices[6]$ be the initial matrix S and restart the iteration process.

C. Calculation of probability weight

To calculate the probability weight, it makes a transition from operation to the input matrix, and starts a process to execute the quantum program. Then it reads ket data before the measurement operation and returns the oracle's probability, denoted by the probability weight for the sensitive value of the quantum register. In Algorithm 2, we show how to make a transition from the recorded operations to the matrices we want, and how to weight these matrices.

Algorithm 2 Weight_Analysis(Operation[], desired[], n)

```
   count=0
2: for Test=1 to Test_time do
     Using (q=Qubit[n])
4:   for i=1 to n do
       Set(q[i], Zero)
6:     for j=1 to Operation.totalnumber() do
         if Operation[j]==Operator then
8:         Operator(q[i])
           //take H gate for example, if Operation[j]==H then H(q[i])
     if Measure(q)==desired then
10:    count++
   return count/Test_Time
```

IV. Preliminary Evaluation

We implement `QuanFuzz` on $Q|SI\rangle$ and $Nrefactory$, for quantum program simulation and code instrumentation respectively, and run on 7 quantum programs with different registers (containing qubits from 2 to 8). These programs are built-in benchmarks when releasing Q$|SI\rangle$. Because we are the first attempt for test case generation of quantum program, there is no related work for comparison. We implement a traditional matrix generator Tradifuzz (traditional branch-coverage guided without quantum sensitive information, similar to Evosuite[])

for comparison. We run both test case generators on each quantum program 5 times and average the results to avoid random factors. The evaluation is performed on a computer with Windows 10 as host OS, Intel i5-4200h as CPU, 16GB of memory. We set the number of the preserved matrices to 6 with the desired probability threshold for a sensitive branch $p = 0.5$ (you can set your preferred number according to the available computing and storage resource). For the `QuanFuzz` execution, when the probability weight for the sensitive value of quantum register reaches the threshold, the iteration stops. For the traditional version, we execute for the same time, and collect the highest weight probability for the sensitive value. Detail results are presented in Table 1.

TABLE I
EXPERIMENT RESULTS USING DIFFERENT GENERATOR

Benchmark	QB_01	QB_02	QB_03	QB_04	QB_05	QB_06	QB_07
Qubit number	2	3	4	5	6	7	8
Iteration	1.2	4.4	4	5.2	5.8	6.2	6.5
Time/s	1.39	48.3	60.9	199.1	595.2	4358.1	10073.2
Pro. of Quanfuzz	0.8	0.748	0.634	0.574	0.7	0.58	0.571
Pro. of Tradifuzz	0.538	0.535	0.328	0.222	0.107	0.056	0.070

From the results, we can see that in all test programs, `QuanFuzz` performs better and successfully triggers the sensitive branch. Besides, it can be seen that with very few iterations (6 qubits register only iterates 5.8 times on average), the probability to trigger the sensitive branch increases significantly, and the coverage of code or bug detection would also increase. But for the traditional input matrix generator, with the same time, the probability to trigger the sensitive branch is only 0.056. We found that for the programs with more quantum bits, it is harder to trigger those sensitive branches with traditional branch coverage guided search-based test case generation techniques, while `QuanFuzz` remains its efficiency. Then, the bugs contained in the sensitive branch can be detected and the whole coverage of these 8 quantum programs can be improved by 20%-60%.

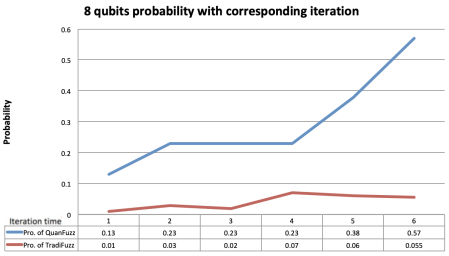

Fig. 4. The probability to trigger the quantum sensitive branch.

based on the sensitive information, `QuanFuzz` always keeps

To better understand the difference between the behavior of `QuanFuzz` and the traditional techniques, we demonstrate the results of QB_07, a quantum program with 8 qubits. As in Fig.3, the traditional version has little improvements in triggering those quantum sensitive branches. Since the traditional input matrix generator cannot understand the behavior of quantum program, it uses the traditional branch based selection such as in EM-Fuzz[12], the effectiveness of which is almost the same as random selection in quantum program. However, getting better matrices in every iteration to increase the probability weight for the sensitive value (i.e value 00101 of qureg q[5] in the motivation example).

Discussion: The time efficiency of the current version is not as good as we thought. Although `QuanFuzz` is fast for programs with low qubit numbers, but it quickly slows down with the accumulation of the qubit number. The time cost is mainly because of the simulation time in current execution platform. We need to simulate the states of those quantum registers in the traditional computer architecture. If we deploy `QuanFuzz` on quantum computer, the time efficiency would be solved. Another issue is the search efficiency of the proposed algorithm. Currently, we use genetic algorithm to select those matrix with higher probability weight for a value of quantum register. More advanced techniques such as static analysis based fuzzing [13] could be customized.

V. CONCLUSION

In this paper, we present `QuanFuzz`, the first attempt for automatically fuzz testing of quantum program. The main idea is to use the search-based algorithm to literately generate unitary gate based matrices to trigger those quantum sensitive branches. `QuanFuzz` obtains 20%-60% more branch coverage than traditional test models to test quantum program. The preliminary results demonstrate its potential use to expose the incorrect behavior of quantum programs at an early stage.

REFERENCES

[1] Lisa Zyga. D-wave sells first commercial quantum computer. https://phys.org/news/2011-06-d-wave-commercial-quantum.html. Accessed June 1, 2011.

[2] Bernhard Omer. A procedural formalism for quantum computing. Master's thesis, Technical University of Vienna, 1998.

[3] Selinger P. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004.

[4] Wang X Liu S, Guan J Zhou L, Duan R Li Y, and Ying M. Q*jSIi*: A quantum programming environment. *arXiv:1710.09500*, 2017.

[5] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: Enabling scalable quantum computing and development with a high-level dsl. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, RWDSL2018, pages 7:1–7:10, New York, NY, USA, 2018. ACM.

[6] R. P. Feynman. Quantum mechanical computers. In *Optics News*, volume 11, page 11, 1985.

[7] Hahn E. M. Feng Y and Zhang L Turrini A. Qpmc: A model checker for quantum programs and protocols. In *Twentieth international symposium of the Formal Methods Europe association (FM)*, 2015.

[8] Yuanliang Chen and Yu Jiang. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th fUSENIXg Security Symposium*, pages 1967–1983, 2019.

[9] Jie Liang and Mingzhe Wang. Fuzz testing in practice: Obstacles and solutions. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*, pages 562–566. IEEE, 2018.

[10] Rick D. Craig and Stefan P. Jaskiel. *Quantum Computation and Quantum Information*. Artech House, 2002.

[11] Aharonov Dorit. A simple proof that toffoli and hadamard are quantum universal. *arXiv:quant-ph/0301040*, 2003.

[12] Jian Gao and Yiwen Xu. Em-fuzz: Augmented firmware fuzzing via memory checking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3420–3432, 2020.

[13] Zhengxiong Luo and Feilong Zuo. Polar: Function code aware fuzz testing of ics protocol. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–22, 2019.