A Preliminary Study on Generating Well-Formed Q# Quantum Programs for Fuzz Testing

Miguel Trinca
INESC-ID & IST, University of Lisbon

João F. Ferreira
INESC-ID & IST, University of Lisbon

Rui Abreu
INESC-ID & FEUP, University of Porto

***Abstract*—Generative Sequence-To-Sequence models have been proposed for the task of generating well-formed programs, an important task for fuzz testing tools such as compilers. In this paper, we propose a Sequence-to-Sequence model to generate well-formed Q# Quantum programs. The ratio of syntactically valid programs among 1,000 Q# files generated by our model is 79.6%. In addition, we also contribute with a dataset of 1,723 Q# files taken from publicly available repositories on GitHub, which can be used by the growing community of Quantum Software Engineering.**

***Index Terms*—Quantum Software Engineering, Fuzz Testing, Sequence-to-Sequence Models, Machine Learning**

I. Introduction

Quantum Computing represents a paradigm shift in computation and promises to deliver a huge leap forward in computational problem solving. This paradigm shift requires not only new programming methodologies and new programming languages, but also reliable and mature tooling that supports these. Compilers, for instance, which transform quantum programs into quantum circuits that can be executed by quantum devices, are an important part of the puzzle. In recent years, several compilers have been proposed by major players in the field, such as Google's Cirq [1], Microsoft's QDK [2], [3] and IBM's Qiskit [4].

It is vital that quantum compilers are reliable, so that the original intentions of quantum programmers are reflected in the quantum circuits that will be executed. However, it is known that compilers may have issues; for instance, during compilation, internal compiler errors (errors intrinsic to the compiler) may be raised. A common practice to identify these issues is to create compiler test suites. According to Chen et al. [5], one of the main challenges is that compilers have multiple options and features, such as different optimizations levels. Thus, it is difficult to create test programs that target specific optimizations levels of the compiler. Another challenge is that these test suites are often manually devised which improves the testing coverage but it also takes a lot of human effort. Grammar-based fuzzing is an effective fuzzing technique that can alleviate human labor and has been proposed to solve this issue. In particular, generative Sequence-To-Sequence models, first introduced by Sutskever et al. [6], have been proposed for the task of generating well-formed programs, towards fuzz testing compilers. One key example is *DeepFuzz* [7], which was used to generate well-formed C programs towards fuzzing C compilers such as GCC.

This work was supported by national funds through FCT under project UIDB/50021/2020

In this paper, we explore the same approach as DeepFuzz and we propose a Sequence-to-Sequence model to generate well-formed Q# Quantum programs. Our ultimate goal is to use the generated programs to test Q# compilers, but here we focus on presenting preliminary results, where we evaluate the performance of our model by measuring the *Pass Rate*, the ratio between all syntactically correct programs and all generated programs. This metric gives us a better insight on how well the model is generating well-formed programs. In our preliminary experiments, we use the model to generate 1,000 Q# files and we obtain a Pass Rate of 79.6%.

In addition, we also contribute with a dataset of 1,723 Q# files taken from publicly available repositories on GitHub, which can be used by the growing community of Quantum Software Engineering. The dataset and generated files are available at: https://github.com/sr-lab/qsharp-fuzz

II. Overview

Our project uses a neural network model denoted as Sequence-To-Sequence [6], which is known for text generation [8] and machine translation [9]. It was also used on more recent studies such as producing well-formed C programs to fuzz compilers (*DeepFuzz* [7]), and producing PDF files to fuzz the PDF parser (*Learn&Fuzz* [10]).

A. Sequence-to-Sequence Model

The sequence-to-sequence model implements two recurrent neural networks (RNN) known as the *Encoder* and *Decoder*. This model was first proposed by Cho et al. [11], where the former RNN encodes a sequence of symbols into a fixed-length vector representation, while the latter decodes the representation into another sequence of symbols. The authors show that the proposed model can learn semantically and syntactically representation of linguist phrases. Also, the authors of *DeepFuzz* demonstrated that it could also learn a programming language's grammar and generate source code.

1) Recurrent Neural Network: A RNN is a neural network that consists of a hidden state h and an optional output y that operates on a variable-length sequence $X = (x_1, ..., x_T)$. At each timestep t, the hidden state is updated as:

$$h_{\langle t\rangle} = f(h_{\langle t-1\rangle}, x_t)$$

(a) Dataset Statistics

Number of repositories with Q# files	131
Average number of files per repository	13
Max number of files in a repository	321
Min number of files in a repository	1
Total number of Q# files	1,723

(b) Project Workflow

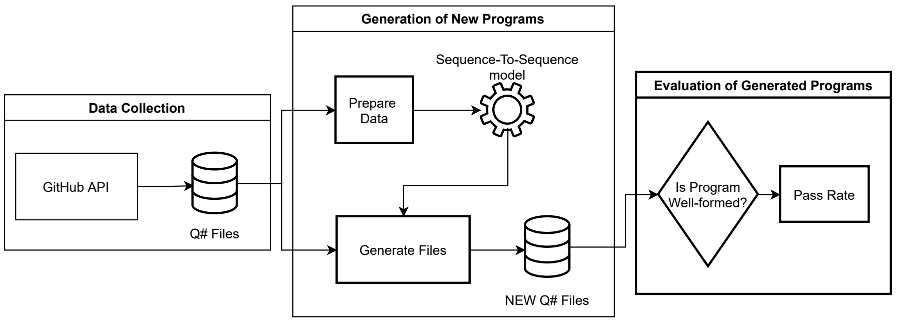

Fig. 1: Dataset statistics and project workflow

Here, f is a non-linear activation function, which can be as complex as a long short-term memory (LSTM) unit.

At each timestep t, the output from the RNN is a conditional distribution $p(x_t|x_{t-1},...,x_1)$, since, by learning to predict the next symbol in a sequence, an RNN can learn a probability distribution. For example, we use a softmax activation function, upon a multinomial distribution of the next character:

$$p(x_t|x_{t-1},...,x_1) = \frac{exp(w_j h_{\langle t\rangle})}{\sum_{j'=1}^{K} exp(w_{j'} h_{\langle t\rangle})}$$

This is defined for all possible symbols $j = 1,...,K$, where w_j are the rows of a weight matrix W. Additionally, we can compute the probability of the sequence X by combining these probabilities:

$$p(X) = \prod_{t=1}^{T} p(x_t|x_{t-1},...,x_1)$$

With the learned distribution, we can iteratively sample new characters at each timestep, thereby generating a new sequence.

2) RNN Encoder-Decoder: This model is composed of two RNNs, the *Encoder*, which learns to encode a variable-length sequence into a fixed-length vector representation, and the *Decoder*, which decodes a given fixed-length vector representation back into a variable-length sequence. In other words, the model is able to learn the conditional distribution over a variable-length sequence X conditioned on yet another variable-length sequence Y, i.e. $p(y_1,...,y_{T'} \mid x_1,...,x_T)$, where T and T' may differ in size. As the *Encoder* reads each character of an input sequence sequentially, the hidden state $h_{\langle t\rangle}$ is updated according to RNN update. In the end, the hidden state is denoted as a summary c of the whole input sequence. On the other hand, the *Decoder* is trained to generate the output sequence by predicting the next symbol y_t, given the hidden state $h_{\langle t\rangle}$. However, the update function of the hidden state is different, since both y_t and $h_{\langle t\rangle}$ are dependent on y_{t-1} and on c:

$$h_{\langle t\rangle} = f(h_{\langle t-1\rangle}, y_{t-1}, c)$$

Similarly, the conditional distribution of the next character is:

$$p(y_t|y_{t-1}, y_{t-2}..., y_1, c) = g(h_{\langle t\rangle}, y_{t-1}, c)$$

Here, f and g are activation functions and g must produce valid probabilities (e.g. softmax). When combining training on both RNNs, we are able to generate a target sequence given an input sequence. Moreover, to maintain information over time, RNNs have feedback loops; in our case, we use LSTMs. LSTM units include a memory cell that can keep information in memory for long periods.

B. Training Data

To train this model, it is crucial to have a significant large dataset with a variety of different code functions. To the best of our knowledge, there is not a publicly available dataset of Q# programs that satisfies our needs. Therefore, we used the GitHub API to search for publicly available GitHub repositories that contain Q# files and we created a dataset with 1,723 Q# files. Once we collected all the the repositories, we searched in each repository for files with the extension *.qs*, which are the Q# files. Besides the source files, we also compiled metadata about the repositories from which these files were taken, such as the *owner name*, *repository name*, and *creation date*. Figure 1a presents some statistics about our dataset.

It is important to note that from the 1,723 Q# files only 244 files were considered for this project, because, in order to compile a Q# program, a *.csproj* must be included. This file details configurations of the compiled program and most of the collected files require file-specific *.csproj* files because they depend on other files, or because they lack the *@Entrypoint* flag (which is the equivalent of a main method in Q#). Therefore, we created a simple *.csproj* specifying only the SDK version and we identified 244 files that can be compiled with it.

C. Workflow

This project is divided into three stages: data collection, generation of new programs, and evaluation of the generated programs. This is depicted in Figure 1b. The first stage, data collection, represents how we collect data towards training the model. This stage has been described in the previous section. Once the files have been collected, we move to the next stage, generation of new programs, where we prepare the Q# programs for training, by creating training sequences, and we train the Sequence-To-Sequence model which has 2 layers with 512 hidden units for each layer. After the model is trained, we use the original Q# programs to generate the new programs. In the third and final stage, we compile the newly 1,000 generated programs in order to compute the *Pass Rate* metric. This metric determines how well the model can generate well-formed files.

III. Design

Looking at Figure 1b we can see that the generation of new programs can be divided into different phases: *Prepare Data, Sequence-To-Sequence model*, and *Generate Files*. In this section, we take a closer look at these phases.

A. Prepare Data

The first step is to remove any noise data and format every Q# file so that it can be split into training sequences. Noise data is in the form of comments and whitespaces, such as horizontal tabs, and newlines. Once this process is done, we split the newly formatted text into sequences with a fixed size, in our case 50 characters. Then, we separate each sequence to be a training sequence, i.e., a pair of input-output sequences, where the corresponding output sequence of an input sequence is the next character of the input sequence. We append every training sequence to a unique file that is used for training. We configure the training to use 80% of the training sequences to limit over-fitting to the whole model.

B. Sequence-To-Sequence Model

Once the *Prepare Data* phase is complete, we configure the training process to learn a generative model over 80% of all training sequences, in order to avoid overfitting.

Before we set up the sequence-to-sequence model and begin training, there are a few steps we need to make. The first one is to vectorize the data. Input and output sequences are separated and every unique character is stored. This is fundamental, since the model can not comprehend characters, and thus the next step is to encode our sequences of characters into sequences of integers. This means that each unique character will be assigned a specific integer value. Then, the sequence-to-sequence model is defined and the training process is ran during 50 epochs.

C. Generate Files

We use the learnt Sequence-to-Sequence model to generate 1,000 new Q# programs. We built a pipeline for accomplishing this endeavor, which starts by picking a file from our dataset. Once a file is chosen, it is divided into three parts. The first part is the *Head*, which is the beginning of the file until the second part is reached. The second part is picked randomly and is a sequence with fixed size, which will be the input for the model. This input is known as *Prefix*. The third and final part is known as *Tail* and it starts after the first *;* is found after the *Prefix*. This means that after the *Prefix* there is some part of the text that is missing. This is what the model is going to try and predict (*Generated*). Thus, the output file can be created as *Head* + *Prefix* + *Generated* + *Tail*. Figure 2b shows an example of a file generated using this strategy, using as a starting point the file shown in Figure 2a. Note that, highlighted in yellow, is what the model predicted that should follow after `Int {`.

IV. Evaluation

In this section we present the experimental setup and the results obtained.

A. Experimental Setup

The Sequence-to-Sequence model had 2 layers and there were 512 LSTM units per layers, with a dropout rate of 0.2. This setup follows closely the same setup of *DeepFuzz* [7], which adopted the same model to generate C programs. We trained the model for 50 epochs on a machine with a 2.40Ghz Intel(R) Xeon(R) Silver 4120R CPU and 64 GB of memory. The model was implemented using Python and the TensorFlow library. In addition, to compile the the Q# files, the .NET Core SDK 3.1 was used.

B. Pass Rate

For the purpose of this preliminary project, the main goal is to evaluate how well the model performs in generating syntactically valid Q# programs. Therefore, we only consider the *Pass Rate*, a metric that measures the ratio of syntactically valid programs among all of the newly generated programs. This metric provides an insight on how well the network has trained over the input sequence.

Our generation strategy greedily predicts the next character and inserts the newly generated sequence based on the same input sequence at one place into the original well-formed program. This prediction uses a model that has been trained over 50 epochs. After collecting the compilation results for each of the 1,000 files, we obtained a *Pass Rate* of 79.6%. This means that out of the 1,000 files only 796 files were syntactically correct. This result is close to *DeepFuzz*, which reports a *Pass Rate* of 82.63% and better than *Learn&Fuzz*, which reports 70%. We argue that the lack of data, and the adopted generation strategy are limitations of this result. While *DeepFuzz* considered 10,000 C files and *Learn&Fuzz* 63,000 non-binary objects, we used only 244 out of the 1,723 Q# files. The model is predicting the likeliest next character after a sequence of characters. It is possible that the sequence that has been given to the model is an entirely new sequence, which the model has not seen yet, due to the lack of data during training. Hence, the model may predict bad characters for this new sequence, creating syntactically incorrect sequences, i.e., syntactically incorrect files. The adopted strategy also has some limitations. First, the predicted sequence of characters may be the exact same of the original file, hence the newly file will be a copy of the original. Second, the model may also

```
namespace QuantumRNG { open Microsoft.Quantum.Canon;
↪   open Microsoft.Quantum.Intrinsic; open
↪   Microsoft.Quantum.Math; open
↪   Microsoft.Quantum.Measurement; open
↪   Microsoft.Quantum.Math; open
↪   Microsoft.Quantum.Convert; operation
↪   GenerateRandomBit() : Result { use q = Qubit() {
↪   H(q); return MResetZ(q); } } operation
↪   SampleRandomNumberInRange(min: Int, max : Int) :
↪   Int { mutable output = 0; repeat { mutable bits =
↪   new Result[0]; for idxBit in 1..BitSizeI(max) { set
↪   bits += [GenerateRandomBit()]; } set output =
↪   ResultArrayAsInt(bits); } until (output >= min and
↪   output <= max); return output; } @EntryPoint()
↪   operation SampleRandomNumber() : Int { let max =
↪   50; let min = 10;  Message($"Sampling a random
↪   number between {min} and {max}: "); return
↪   SampleRandomNumberInRange(min, max); } }
```

(a) Original file

```
namespace QuantumRNG { open Microsoft.Quantum.Canon;
↪   open Microsoft.Quantum.Intrinsic; open
↪   Microsoft.Quantum.Math; open
↪   Microsoft.Quantum.Measurement; open
↪   Microsoft.Quantum.Math; open
↪   Microsoft.Quantum.Convert; operation
↪   GenerateRandomBit() : Result { use q = Qubit() {
↪   H(q); return MResetZ(q); } } operation
↪   SampleRandomNumberInRange(min: Int, max : Int) :
↪   Int { mutable output = 0; repeat { mutable bits =
↪   new Result[0]; for idxBit in 1..BitSizeI(max) { set
↪   bits += [GenerateRandomBit()]; } set output =
↪   ResultArrayAsInt(bits); } until (output >= min and
↪   output <= max); return output; } @EntryPoint()
↪   operation SampleRandomNumber() : Int {
↪    let min = 10;  let min = 10;  Message($"Sampling
↪   a random number between {min} and {max}: "); return
↪   SampleRandomNumberInRange(min, max); } }
```

(b) Generated file

Fig. 2: Examples of two valid Q# files: (a) a file from our dataset; (b) a file generated by our model based on the file shown in (a)

not predict when to close brackets or quotes. An example is when the prefix sequence given to the model has a quote ("): the model keeps predicting other characters and might never close the quote. This will generate a syntax error once the new file has been created. This shows the importance of creating different strategies that overcome these limitations.

V. Related Work

The closest work to our project is *DeepFuzz*, a grammar-based fuzzing tool based on a generative Sequence-To-Sequence model [7]. The goal of DeepFuzz is to produce well-formed C programs towards fuzzing C compilers such as the GCC. Other work has been done in machine translation [9], such as *Learn&Fuzz*, which shows how to automate the generation of PDF files to fuzz the PDF parser, using a Char-RNN Language model [10].

Regarding datasets with quantum programs, Bugs4Q is a benchmark of thirty-six real, manually validated Qiskit bugs [12]. Also, recent work has been done towards QBugs, a novel infrastructure, which will provide a variety of information, from source code, test cases, and bug reports to configuration files and scripts [13]. However, to the best of our knowledge, QBugs is not yet publicly available.

VI. Conclusion and Future Work

We created a new dataset consisting of 1,723 Q# programs, which can be used by the growing community of Quantum Software Engineering. We also implemented and tested a generative Sequence-To-Sequence model to generate well-formed Q# programs. We found that 79.6% of 1,000 files generated were syntactically correct. Despite the limitations already discussed, we believe that the generative Sequence-To-Sequence model proposed can be a viable option to create new Q# programs. In future work, we intend to grow the dataset and implement new generation strategies.

References

[1] Google, "A Python framework for creating, editing, and invoking Noisy Intermediate Scale Quantum (NISQ) circuits.." https://github.com/quantumlib/Cirq, 2021. [Online; accessed 10-April-2021].

[2] Microsoft, "Quantum Development Kit." https://azure.microsoft.com/en-us/resources/development-kit/quantum-computing//, 2021. [Online; accessed 10-April-2021].

[3] K. Svore, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, A. Paz, and M. Roetteler, "Q# enabling scalable quantum computing and development with a high-level dsl," in *Proceedings of the Real World Domain Specific Languages Workshop 2018*, pp. 1–10, 2018.

[4] G. Aleksandrowicz, T. Alexander, P. Barkoutsos, L. Bello, Y. Ben-Haim, D. Bucher, F. Cabrera-Hernández, J. Carballo-Franquis, A. Chen, C. Chen, *et al.*, "Qiskit: An open-source framework for quantum computing," *Accessed on: Mar*, vol. 16, 2019.

[5] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[6] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, 2014.

[7] X. Liu, X. Li, R. Prajapati, and D. Wu, "Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 1044–1051, 2019.

[8] I. Sutskever, J. Martens, and G. E. Hinton, "Generating text with recurrent neural networks," in *ICML*, 2011.

[9] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush, "OpenNMT: Open-source toolkit for neural machine translation," in *Proceedings of ACL 2017, System Demonstrations*, pp. 67–72, 2017.

[10] P. Godefroid, H. Peleg, and R. Singh, "Learn&Fuzz: Machine learning for input fuzzing," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 50–59, IEEE, 2017.

[11] K. Cho, B. van Merrienboer, C. Gulcehre, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," in *Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*, 2014.

[12] P. Zhao, J. Zhao, Z. Miao, and S. Lan, "Bugs4Q: A benchmark of real bugs for quantum programs," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1373–1376, IEEE, 2021.

[13] J. Campos and A. Souto, "Qbugs: A collection of reproducible bugs in quantum algorithms and a supporting infrastructure to enable controlled quantum software testing and debugging experiments," in *2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*, pp. 28–32, IEEE, 2021.