Background

- **Scientific software**: software applications primarily focused on exploration and analysis of data.
- Mostly developed by researchers/graduate students with deadlines, not software developers; potential introduction of bugs.

Motivation

- As researchers write more code, higher probability of errors follow (Soegaard, 2015).
- Traditional software testing methods are not always enough to find critical, result-altering bugs (e.g., Bhandari Neupane et al. 2019).

Traditional software testing methods include:

- **Fuzzing**: feeding predesigned data to a program to trigger crash/unexpected behavior.
- **Mutation testing**: altering inputs, not software prone inputs.
- **Mutation based fuzzing**: use grammars to generate inputs; more likely to find bugs.
- **Random based fuzzing**: purely random strings; less likely to find bugs.

**Research Goal**

The goal of this research project is to help scientists in conducting correct and verifiable scientific computations by identifying latent bugs in scientific software.

**Research Question**

How can we identify crash-prone inputs that can be used as fuzz data to discover bugs in scientific software?

Hypothesis

Through qualitative analysis we can identify characteristics of bugs in scientific software that we can leverage to identify undiscovered bugs.

**Datasets**

- List of fuzzed Julia packages:
  - FFTW.jl
  - GLFW.jl
  - HTTP.jl
  - WebSocket.jl
  - LightXML.jl
  - LinearOperators.jl
  - SymPy.jl
  - MachineLearning.jl
  - NeuralNetwork.jl
  - Random.jl
  - User bullpen 16

**Methodology**

- **Traditional fuzzing**: hand-written fuzzers.
- **Three typical methods**:
  - **True random inputs**
  - **Generation-based**
  - **Mutation-based**

**Methodology (continued)**

- **Public GitHub Julia Repositories**
  - Unify variable naming convention, code style
  - Pre-Processed Julia Files
  - Learn Julia structure/syntax
  - LSTM Generative Model
  - Sample LSTM model results
  - Julia files for testing

**Results**

- **Table 1**: 3 approaches for traditional fuzzing.
  - We first implemented traditional hand-written fuzzers in Python for 7 Julia repositories.
  - Combinations of random, generation-based, and mutation-based fuzzers.
  - **Table 2**: Bugs discovered using traditional fuzzing approach.
  - Created multiple testbeds to run same test.
  - Multiple testbeds help to discover bugs.

**Figure 1**: Result-altering bugs due to different operating systems, reproduced from Bhandari Neupane et al. (2019).

**Figure 2**: Different stages of fuzzing a program.

- Wrote traditional fuzzers for each unique function in each of 7 Julia libraries.
- We then began to implement automated fuzzing techniques utilizing machine learning following Cummins et al. (2018).

- Mined 8,211 public Julia repositories from GitHub (107K Julia files, in 34K directories, ~8.2M lines of code) to obtain training corpus.
- Applied consistent variable naming/code formatting to ease machine learning process.
- Will use Long Short-Term Memory (LSTM) style of Recurrent Neural Network to learn structure and syntax of Julia.
- Sample LSTM to create very large (~1M files) synthetic corpus as a test suite: represents “repositories that have yet to be written”.

**Results**

- Traditional fuzzing has been completed for 7 Julia repositories using Python.
- Traditional approach required over 3 months and does not scale well; a scalable solution is needed.

**References**


**Acknowledgements**

We would like to thank our CSC-6220 professor, Dr. Akond A. Rahman. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. 1649609. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.