Python Predictive Analysis for Bug Detection

(Artifact)

Zhaogui Xu[*], Peng Liu[†], Xiangyu Zhang[†], and Baowen Xu[*]

[*] State Key Laboratory of Novel Software Technology [†] Department of Computer Science
[*] Nanjing University, China [†] Purdue University, USA

zgxu@smail.nju.edu.cn {peng74,xyzhang}@cs.purdue.edu bwxu@nju.edu.cn

1. SCORECARD

1.1 Insightfulness

Python is one of the most popular programming languages nowadays. It has been used in a wide range of areas including but not limited to web applications (e.g., Django, Flask and Requests), machine learning frameworks (e.g., Sklearn and TensorFlow), natural language processing (e.g., NLTK), remote management (e.g., Salt and Fabric) and cloud computing (e.g., OpenStack). Most recently, big data scientists are extensively applying deep learning frameworks backed by Python to make unprecedented innovations such as cancer prediction, face recognition, and unmanned vehicles.

Despite its flexibility, Python is a dynamic language. The scientists and practitioners that are using Python also risk introducing type errors to their systems. These type errors may cause further exceptions or produce incorrect results. Due to the prominence and the prevalence of the Python applications, it is a demanding task to detect type errors before the deployment of any Python program. Unfortunately, very few automated tools exist despite the developers' desperate need. Our tool is designed and developed to fill the gap.

Automated type error detection tools are largely lacking because of the unique challenge imposed by the dynamic type system of Python. For example, a variable can hold different types of values at runtime depending on the inputs and the execution paths. Therefore, a bug finding tool needs to find the specific input and the execution path that trigger the error. Traditional analyses do not apply because they assume static typing and cannot handle the dynamic features of Python. Our novel design sheds light on how to model dynamic features in symbolic analysis. Moreover, our engine leverages a constraint solver to precisely pinpoint when and how a type error occurs.

More specifically, we develop the first predictive symbolic analysis engine to identify bugs and their triggering inputs in Python programs. Instead of exhaustively exploring every program path, our technique aims to explore neighboring executions of an observed execution by relaxing the inputs with different types, values or attribute sets. We design a novel encoding scheme for Python programs, which focuses on handling dynamic features by introducing symbolic variables to represent dynamic types, attributes and explicit reasoning about their changes and correlations. We leverage a solver to implicitly explore neighboring executions. In fact, our technique can be used to detect various kinds of bugs that can be expressed in the form of assertion violations.

1.2 Usefulness

Our tool is highly useful in practice. It can detect bugs in Python programs that involve dynamic features and hence are difficult for existing techniques. This is evidenced by the fact that we have successfully detected 46 bugs (16 new) in 11 popular Python programs. Most of our reported bugs are confirmed and then fixed by the developers. Besides, we also provide the real issue identifiers of these bugs and some detailed case studies to facilitate future research.

Our tool is cost-effective. It can detect most of these bugs in a few seconds to a few minutes. Note that lots of these bugs require long execution paths to trigger bug and involve complex business logics that are difficult to understand, indicating that the developers can hardly find them without the support of our automated tool.

1.3 Usability

Is it easy to understand? Our paper has provided detailed description about the technique and the experiment. In the artifact package, we have also prepared detailed description about how to use the tool and how to validate the results.

Is it accompanied by tutorial notes? We have prepared a README file that includes a detailed tutorial.

Is it easy to download, install, or execute? We put our system, benchmarks, configurations, running scripts and documentation on a virtual machine so that (1) the execution environment is properly set up a priori to avoid tedious configuration and compatibility issues, and (2) execution inside a sandbox will not affect the host machine. The VM image can be downloaded from the project website `https://sites.google.com/site/pypredictor/`. For a quick start, we have installed our tool, all the benchmarks and the required environments on the VM image. To start the tool, the users only need to input several commands in the terminal. For easy validation, the results generated by our tool are also pretty printed. The last but not least is that we make the project source code publicly available. It can also be downloaded from the project website `https://sites.google.com/site/pypredictor/`.

2. ARTIFACT DESCRIPTION

We prepare our artifact package on a VMWare image which can be started on any host with VMWare Workstation (version 10) installed. In this section, we will give detailed description about how to use the system.

Installation of VMWare Workstation. Our artifact image requires VMWare Workstation 10, which can be downloaded from [1]. Note that the Linux version of VMWare Workstation is packaged in a "`*.bundle`" executable so that one can simply use the command "`sudo sh /path/to/<filename>.bundle`" to start installation.

Downloading and Loading the VM Image. First of all, go to the project website `https://sites.google.com/site/pypredictor/` to download the VM image. After decompressing it, please start VMware Workstation and go to menu `File>Open>Select the *.vmx file` to load the image.

Artifact Contents in the Image. After loading the image, the HOME directory contains the following contents:

- A README file on the desktop (i.e., `$HOME/Desktop`), which includes the requirements, instructions, evaluation and so on.
- A virtual environment (`$HOME/Pythons/virtualenvs`) of Python with our tool and all the benchmarks installed.
- A working directory (`$HOME/Desktop/testcases`), including all the test cases, configurations and running scripts.
- A mapping file `issue-id-mapping.txt` on the desktop, describing the mapping between test cases and the corresponding issue identifiers.

Source Code and Case Studies. We make the source code of our tool publicly available. It can be downloaded from the project website `https://sites.google.com/site/pypredictor/`. Our tool can be installed on Python 2.7 (Linux). We also provide basic tutorials about how to install and run the system through a small buggy program. For a better study of our technique and system, we also put some case studies on the project website `https://sites.google.com/site/pypredictor/` where we describe in details how bugs occur, how our technique detect them and the analysis results of our tool.

Running the Tool. Before running the tool, please make sure the **network** of the VM works properly. Otherwise, the user may not get the right results of some cases. To start the tool, please follow the steps below:

1. Open a terminal.
2. Switch to the Python virtual environment by: `source $HOME/Pythons/virtualenvs/python2.7.1/activate`.
3. Move to the working directory by command: `cd $HOME/Desktop/testcases`.
4. Configure the Salt project by: `./saltconf.sh`. Test the configuration by: `salt "*" test.ping`. If the test prints "True", the configuration should be good.
5. For a quick and full evaluation, input the command: `./run.sh`. It will take 30-40 minutes to complete.
6. To evaluate each case one by one, input the command: `python test_caseX.py` where `X` means the number of the test case.

We also provide the above instructions in the README file.

Evaluating the results. The evaluation goal is to reproduce the results shown in Table 3, which represents our main results. We have three kinds of output files during the execution of our tool. All of them are located in folder `testcases/output` by default. They are listed as follows:

- `TraceMatrix_test_caseX.txt` stores the detailed statistics of tracing results for test case `X`. Refer to README for the meaning of each record.
- `PredictMatrix_test_caseX.txt` stores the detailed results of the predictive case. Also refer to README for the meaning of each record.
- `PyTracer_test_caseX.db` stores the dumped trace of test case `X`.

Refer to the issue mapping file `issue-id-mapping.txt` to align the results to our paper. For more details about the evaluation, please refer to the Section [Evaluation] in the README file.

3. REFERENCES

[1] VMWare Workstation
https://my.vmware.com/web/vmware/info?slug=desktop_end_user_computing/vmware_workstation/10_0