FuzzDiscover: A Fuzz Experiment Generator for ChaosToolkit

Anjali Menon, *University of Illinois at Urbana-Champaign*

Abstract

Since its inception in late 2000s, the concept of chaos engineering has been directed only towards infrastructure failure. In this paper, we extend the scope of chaos engineering to include chaos in application input. We propose a fuzz chaos experiment generation extension, FuzzDiscover [1], for the opensource chaos engineering tool ChaosToolkit [2]. FuzzDiscover generates contextualized black-boxed fuzz tests and metrics that monitor degradation in system state in terms of CPU, memory, disk space, database connection pool utilization and process count. FuzzDiscover provides chaos experiments to ChaosToolkit for running and checks if the steady-state hypothesis as defined by these metrics is met at the end of the chaos experiment.

1. Introduction

Netflix introduced the concept of Chaos Engineering to the world in 2008 in through the white paper [3] listing engineering steps for testing large-scale infrastructure failures. Over the years multiple opensource and proprietary software have been developed that have furthered the scope of chaos testing to include a wide array of network and infrastructure failures. However, the concept of "chaos" affecting the system need not be limited to network and infrastructure alone. Bad or random input is another form of chaos that a system needs to be able to defend itself against.

The expectation from a software system when it is provided an incorrect input is that it will cause the execution of the particular request alone to fail. However, poorly written programs can cause other abnormal behaviors as a side effect of unexpected input. They can be indications of unintentional fork bombs or memory, disk space or database connection leaks. These abnormal behaviors, that degrade the systems health by using up system resources long after the failed input execution has finished, can have affect the execution of future requests or other processes running in the system.

In this paper, we propose a tool for generating fuzzed inputs for an application that attempt to surface system-health degrading behaviors. The tool also looks at metrics that indicate if such behaviors have occurred. Currently the metrics checked include CPU, memory, disk space, database connection pool utilization and process count. Any deviation in these values at the end of the experiment from its initial ones, within certain tolerance limits, are taken as an indication that the inputs we have run potentially degrade the system's health.

This tool generates chaos experiments for ChaosToolkit, that perform their sequential execution and rollback instructions that FuzzDiscover provides to it. For the generation of fuzzed data, FuzzDiscover uses a black-boxed fuzzer, kittyfuzzer [4].

1.1. Experiment generation for ChaosToolkit

ChaosToolkit, on a high level, accepts as inputs the scripts to run for testing, checking steady-state hypothesis metrics and performing rollback. The tool is fully reliant on user-input and only performs the execution of scripts it is explicitly provided.

FuzzDiscover provides these scripts to ChaosToolkit in the JSON structure it can read. For steady-state hypothesis checking, FuzzDiscover provides ChaosToolkit a single Python script it can execute once before and once after the experiment. For rollback, FuzzDiscover gives Chaostoolkit explicit commands for temporary file deletion and, when required, restoring application files by copying over from FuzzDiscover's backup. The tests for running are provided as multiple scripts for different fuzzed inputs

1.2. Fuzz generation from kittyfuzzer

Kittyfuzzer provides fuzzed data from the data modal that is provided to it. FuzzDiscover generates the data modal by analyzing sample user input and asks kittyfuzzer to generate fuzz for explicit sections of the input.

The derivation of a data modal and the fuzzing of only certain portions of it are to ensure that only meaningful fuzzes are generated. For instance, if an input to an application looks like "curl -k https://localhost:8443/city/Chicago", then it is intuitive that only certain parts of the input need to be fuzzed, namely, "https" can be replaced with other protocols such as "ftp", "sftp" etc., "localhost", "city", and "Chicago" can be replaced with random character strings, and "8443" can be replaced with a random number.

2. Software Features

FuzzDiscover can fuzz two "areas" that serve as inputs to an application:

1. User input, which is the explicit command that is run for executing the application.
2. Input files, which are files that will be read by the application.

```
{
 "start_up" : [],
"sample_input" : [],
 "fuzz_internal_files_path_to_sources" : [],
 "input_files" : [],
“is_annotated” : “true/false”
}
```

Figure 1: Input structure for FuzzDiscover

```
{
 "steady-state-hypothesis" : {
      "probes": []                    },
"method" : [],
 "rollbacks" : []
}
```

Figure 2: Output structure for FuzzDiscover/ Input structure for ChaosToolkit

For fuzzing user input, user needs to provide a sample input to fuzz discover. The user can choose to fuzz only certain parts of the input by providing explicit annotations about which parts to fuzz and how using FuzzDiscover's annotations. FuzzDiscover supports the following annotations: @@{ct_Fuzz_Number{ }} and @@{ct_Fuzz_String{ }} which fuzzes the sections enclosed within the parenthesis with fuzzed numbers or character strings respectively.

For fuzzing input files, user can choose to explicitly specify the paths of the input files to fuzz or ask FuzzDiscover to automatically fuzz any file that will be read by the application. For the latter, which we refer to as internal file reads, the paths to the source code files of the applications must be provided by the user.

For checking if the system meets the steady-state hypothesis after the experiment FuzzDiscover checks for CPU, memory, disk space, and database connection pool utilization before and after the system. FuzzDiscover mandates that these metrics should fall within the following tolerance limits for steady state to be met:

1. CPU Utilization:
   - The final CPU idle time after the experiment should not be less than 50% of its initial value before the beginning of the experiment.
   - The final CPU utilization should not be greater than 97%.
2. Memory utilization:
   - The available unused memory after the experiment should not be less than 50% of its initial value before the beginning of the experiment.
3. Disk space utilization:
   - The final available free disk space after the experiment should not 50 MB lesser than its initial value before the beginning of the experiment.
4. Process count:
   - The process count in the system should not be greater than 1.5 times the number of processes in the system at the starting of the experiment.
5. Database connection pool count:
   - The number of available database connections should remain the same before and after the experiment.
   - There should be connections readily available to the database both before and after the experiment, i.e., the connection pool must not be full.

Currently FuzzDiscover only performs the database connection checks for MySQL databases and requires username and password for accessing the database to be explicitly provided.

3. Software Design

FuzzDiscover accepts as input a JSON file [Figure 1] with the following fields: fuzz_internal_files_path_to_sources, start_up, sample_input, input_files and is_annotated and produces another JSON file [Figure 2] containing the experiment details for ChaosToolkit to run.

The output experiment JSON as accepted by Chaostoolkit contains three fields:

1. **method**: a set of scripts that constitute the actual test runs.
2. **steady-state-hypothesis**: a set of probes or metrics to check once before and once after the experiment that assert correctness, which in our case is that system state has not deviated.
3. **rollbacks**: a set of scripts that will be executed once after completion of the experiment irrespective of the experiment's outcome.

```
{
 "start_up" :  ["python3 sunset.py", "python3 astre.py"],
"sample_input" :
["curl -k https://localhost: @@{ct_Fuzz_Number{8443}} / @@{ct_Fuzz_String{city}} / @@{ct_Fuzz_String{Chicago}} "],
 "is_annotated" : "true"
}
```

Figure 3: Example of Annotated Input for FuzzDiscover

3.1. Generating method:

The type of methods generated by FuzzDiscover are determined by the area of fuzzing specified by the user. In all cases a data modal for the input is provided to kittyfuzzer which writes each fuzz it generates into a separate file.

For user input fuzzing, these generated fuzz files are provided as executable scripts in method. Special commands are added within method to provide executable permissions for these files. For input file fuzzing, explicit commands are provided in method for copying over the contents of fuzzed files to the target input file.

3.1.1. User Input Fuzzing

FuzzDiscover parses the sample input and determines the data modal for fuzzing for kittyfuzzer.

3.1.1.1. With user annotation

When user has specified the sections to be fuzzed through annotations in sample_input, only those sections are marked for fuzzing in the data modal.

3.1.1.2. Without user annotation

When no annotations have been given by the user, FuzzDiscover attempts to guess the correct data modal by parsing sample_input and looking for certain regexes and substrings.

- FuzzDiscover finds substrings that are numbers or character strings and marks them for appropriate fuzzing in the data modal.
- All special characters are marked as constants and are not fuzzed.
- When the input is delimited by space, FuzzDiscover guesses that the first section of the input is the original command to be run such as “curl” and does not fuzz it.
- Special substrings present in sample_input which are network protocols such as “http”, “https”, “ftp”, “sftp”, “ftps” and “ssh”, are only replaced in the fuzzed data with other network protocol. Similar rule is followed for http headers such as “GET”, “HEAD”, “POST”, “PUT”, “DELETE”, “TRACE”, “OPTIONS”, “CONNECT”, and “PATCH”

```
import sys
sys.setrecursionlimit(1 << 30)
def fibo(num):
    if num == 1 or num == 0:
        return num
    return fibo(num - 1) + fibo(num - 2)
store_file = "cs598_fibo_demo_internal_file.txt"
with open ( store_file , 'w') as __w_file:
    for i in range(len(sys.argv) - 1):
        __w_file.write(sys.argv[i + 1] + "\n")
with open ( store_file , ‘r’ ) as __r_file:
    for __line in __r_file:
        print(fibo(int(__line)))
```

Figure 4: Source code for target nth Fibonacci number generator number program

3.1.2. Input file fuzzing

3.1.2.1. User provided input file

FuzzDiscover stores a backup of the original contents of the files given in input_files. The contents of fuzz files generated by kittyfuzzer are copied over to these files in input_file before each test execution of the application in the experiment.

3.1.2.2. Internal-file-read fuzzing

FuzzDiscover stores a backup of the original contents of the source files in fuzz_internal_files_path_to_sources. FuzzDiscover then parses these source files and identify substrings matching regexes for file reads. (These regexes are specifically hard-coded for different programming languages. Currently FuzzDiscover supports only Python for this feature.) The source code is instrumented to replace these substrings with reads to “fuzzed_file.txt”. For instance, a section of the source code which is “open(a_file,‘r’)” would be replaced with “open(‘fuzzed_file.txt’,‘r’)“. The fuzz files generated by kittyfuzzer are then copied over to “fuzzed_file.txt” before each test execution of the application in the experiment in the same manner as in section 3.1.2.1.

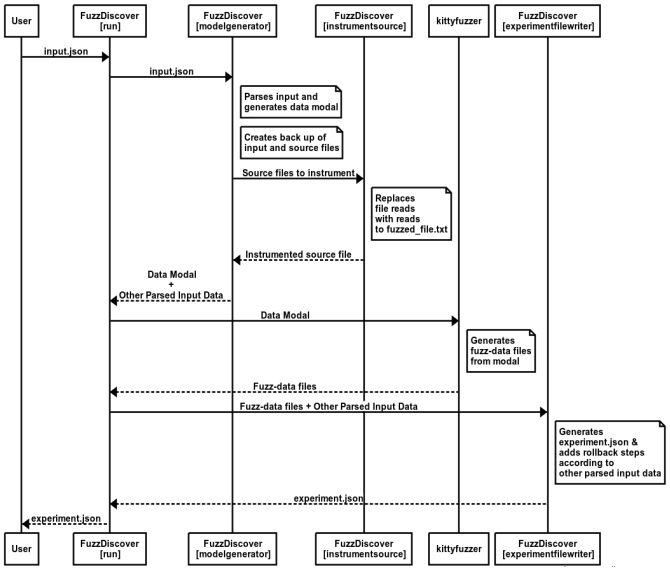

Figure 5: Sequence diagram for Chaostoolkit experiment JSON file generation by FuzzDiscover

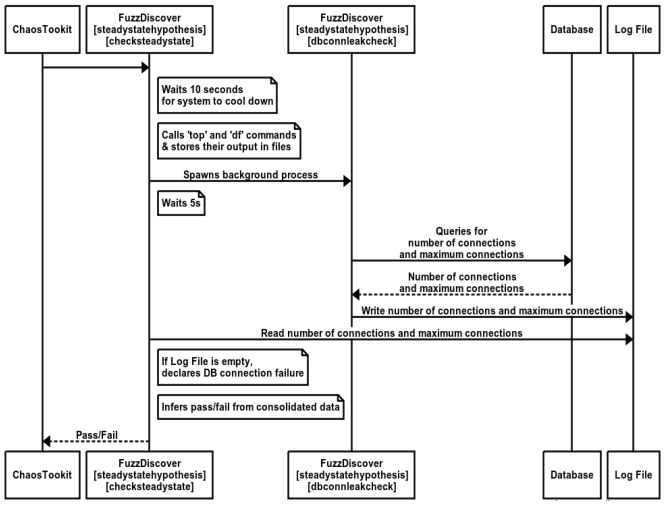

Figure 6: Sequence diagram for steady-state-hypothesis verification through FuzzDiscover

3.2. Generating steady-state-hypothesis:

The steady-state-hypothesis that is provided to ChaosToolkit by FuzzDiscover is a single Python script which contains the consolidated checking of the metrics specified in section 2. To discount any momentary spikes in values due to the experiment, a 10 second sleep is given between the completion of test executions and the final sampling.

CPU idle time, unused memory, and process count are process count are obtained from the command-line 'top' command. At both initial and final checks of the steady state hypothesis, 10 consecutive samples are taken at 1 second intervals and their averages are used for the comparison. This is to account for momentary spikes in their values. Disk space for each disk is similarly obtained using the "df" command, but with only one sample taken at each check (since disk space values rarely show momentary spikes). Disk space value checks are performed after experiment data generation and before cleanup to discount for the presence of FuzzDiscover's temporary files affecting disk utilization.

The current number of database connections, and maximum connections in the pool are obtained by querying the MySQL database with the queries "select count(*) as current_conn_count from processlist" and "SHOW VARIABLES LIKE 'max_connections'" respectively. In the case where the connection pool has already been exhausted at the time of our querying, our program would remain suspended for a long time waiting to obtain a connection. In order to avoid this, our steady-state checking program instead spawns off a new process for performing the database querying. The parent checking program waits for 5 seconds for the child process to finish execution and write its findings into a log file. At the end of 5 seconds, if the parent does not see logs from the spawned child, it assumes that the child is still waiting for connection to the database and fails the steady-state-hypothesis check citing a database connection pool exhaustion.

3.3. Generating rollbacks:

User provided commands in start_up along with commands for cleaning up fuzz and temporary files generated by FuzzDiscover are provided as rollback steps to ChaosToolkit. Additionally, in case of input file or internal-file-read fuzzing, the respective input and source files are restored from the backup FuzzDiscover had created prior to the starting of the experiment.

4. Experimental Demonstration

We evaluate FuzzDiscover by demonstrating its usefulness in uncovering potential for degrading system state in a Python program. We choose the target of demonstration as a simple program which computes nth Fibonacci numbers through recursion, whose source code is shown in Figure 4.

Results: On running the internal-file-read fuzzing experiments generated by FuzzDiscover, there was a failure in meeting the steady-state-hypothesis due to abnormal decrease in disk space. This was due to segmentation faults caused by some very large fuzz data that were read by the now-instrumented code. Segmentation faults and memory leaks can cause core dumps in the system. Core dumps are large files that slowly eat away disk space with each failed execution and their creation often goes unnoticed until the

strains on disk space availability are large enough to be perceptible. In this execution, the segmentation fault resulted in dumping of core files each of around a 100 MB, and the change is disk space utilization was caught by FuzzDiscover's steady-state checker.

5. Discussion and Future Work

FuzzDiscover relies on black-box fuzz data generation and therefore cannot generate sufficiently "smart" test data. The current attempts by FuzzDiscover to contextualize the input and generate data modal are rudimentary. There is a significant scope for improving the intelligence of the fuzz data generation.

FuzzDiscover performs blind copy during its file backup, and for large files can add non-trivial overhead to disk space. The source code instrumentation performs line by line reading of the file and therefore in addition to being non-optimized for performance can lead to potential out of memory failures for extremely large lines.

Also, source code instrumentation is based on regex checking for known ways of file reads. It would not be able to identify and perform instrumentation when less known or unconventional file access methods are used.

The thresholds for checking steady-state deviations are intuitively sane but lack experimental verification. The current threshold intervals can be too wide and allow relevant failures to pass through. Similarly, if the system is already under heavy load at the start of the experiment, the intervals can be too narrow and trigger steady-state check failures for momentary and irrelevant spikes in the respective values. The values can also be affected by other processes running on the same machine.

Additionally, FuzzDiscover currently uses only a handful of basic metrics for monitoring system health and does not check for other potential resource contention issues.

Moreover, fuzzing as a technique is best suited for surfacing security-related issues. Its scope for uncovering system degrading issues is limited. Better techniques need to be found that would more easily trigger such issues.

6. Related Work

There are numerous chaos engineering tools and frameworks available today, with some of the more popular ones such as Chaos Monkey [5], and Gremlin [6] being used by tech-giants such as Netflix, Amazon and Twilio. All other major tech-companies also employ chaos engineering principles in one form or the other for testing their distributed system.

Similarly, there are numerous fault-injection frameworks, including intelligent fuzz testing tools such as American Fuzzy Lop [7], Hodor [8] and HonggFuzz[9]. However, they attempt to generate inputs that fail the execution of a single request as opposed to regrade system health. FuzzDiscover is one of the first tools to explicitly combine the concepts of chaos engineering with non-network/infrastructure fault injection.

7. Conclusion

FuzzDiscover builds upon the principles of chaos engineering and find a new approach for breaking software systems. While the current tool relies on elementary techniques for surfacing system degrading behavior in applications, its demonstrability in seemingly innocuous and simple applications proves its usefulness.

Acknowledgments

I would like to thank Professor Tianyin Xu for his guidance and helpful feedback. The insights that aided in the conceptualization and implementation of this tool would not have existed without Suneel Gelli, Sanil Mohandas and Aravindh Ramaswamy of Amazon Development Centre Chennai.

References

[1] https://github.com/anjalim3/chaostoolkit-fuzzdiscover

[2] https://chaostoolkit.org/

[3] https://principlesofchaos.org/

[4] https://pypi.org/project/kittyfuzzer/

[6] https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey

[6] https://www.gremlin.com/

[7] http://lcamtuf.coredump.cx/afl/

[8] https://github.com/nccgroup/Hodor

[9] https://github.com/google/honggfuzz

[10] https://www.techrepublic.com/article/chaos-engineering-a-cheat-sheet/

[11] https://www.usenix.org/legacy/event/lisa07/tech/full_papers/hamilton/hamilton_html/

[12] https://github.com/bouncestorage/chaos-http-proxy

[13] https://github.com/dastergon/awesome-chaos-engineering

[14] https://stackoverflow.com/questions/6470651/creating-a-memory-leak-with-java

[15] ftp://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz.pdf

[16] https://people.csail.mit.edu/rinard/paper/icse09.pdf

[17] https://blackarch.org/fuzzer.html

[18] https://github.com/bbc/chaos-lambda

[19] https://github.com/Optum/ChaoSlingr

[20] https://github.com/osrg/namazu

[21] https://github.com/alexei-led/pumba