NVME Testing and Logging Report

David Kang

8/29/2022

Introduction

The goal of this project was to improve the testing of SSDs, and one way to do that was by automating the process. SSDs, before being put out into the market, are tested for errors by semi-manually running commands through them. One type of such commands are NVMe commands that run on NVMe (non-volatile memory express) SSDs and can be run to test the SSD.

An open-source NVMe interface tool emulator in C was found called nvme-cli on GitHub (https://github.com/linux-nvme/nvme-cli). Using that, I created automated scripts that ran commands with different inputs systematically to check for valid vs. invalid inputs and for errors.

At first, fuzzing and using AFL++ was considered for automation. However, AFL++ is better suited for black box fuzzing, and we wanted to use model-based fuzzing, so we did not use it. Instead, I wrote my own script to generate parameters for NVMe commands.

This report describes how I set up, the internals of the test script, and how I ran the tests.

System

Computer Hardware

Dell Inc. OptiPlex 7040
Memory: 8 GB
Processor: Intel Core i5-6500 CPU @ 3.2GHz x 4

SSD

Kingston KC3000, PCIe 4.0 NVMe M.2, 512 GB

Operating System

Ubuntu Desktop

Software Setup

Install nvme-cli

To install original nvme-cli from GitHub (may be updated and different from version used):

```
device-name:~$ git clone https://github.com/linux-nvme/nvme-cli
```

To install nvme-cli edited version of nvme-cli (instrumented and including logger code):

```
device-name:~$ git clone https://github.com/dkminsoo/ssd-testing
```

Compiling nvme-cli

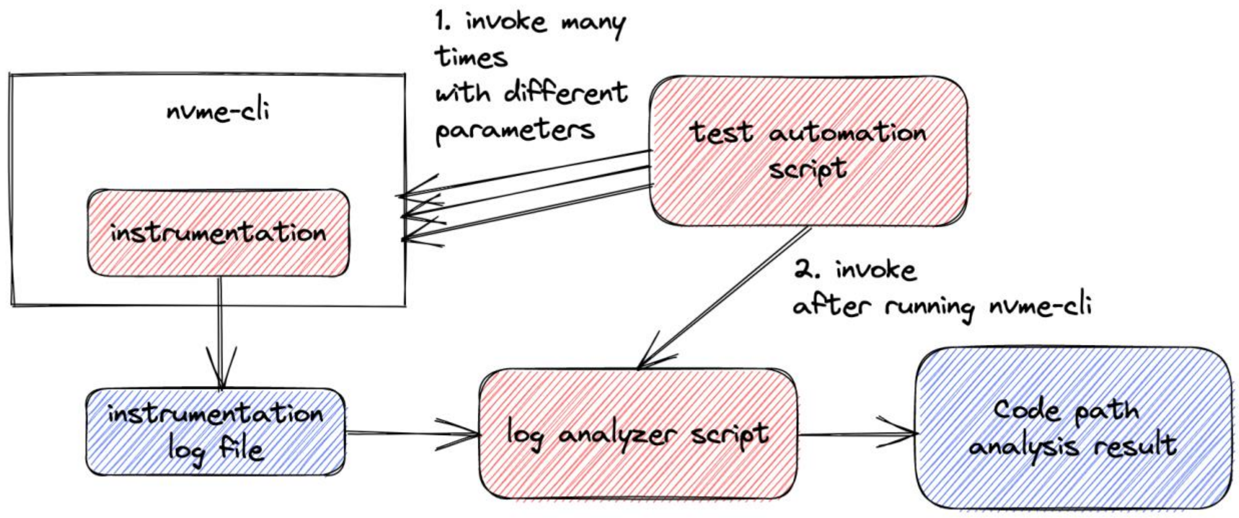
To compile nvme-cli, I had to install the following additional packages:

```
device-name:~$ sudo apt-get update
device-name:~$ cd nvme-cli/
device-name:~/nvme-cli$ sudo apt install pkgconf
device-name:~/nvme-cli$ sudo apt install libpam0g-dev
device-name:~/nvme-cli$ sudo apt install uuid-dev
device-name:~/nvme-cli$ sudo apt install libcryptsetup-dev
device-name:~/nvme-cli$ sudo apt install meson
device-name:~/nvme-cli$ meson .build
device-name:~/nvme-cli$ ninja –C .build
device-name:~/nvme-cli$ meson install –C .build
```

If changes are ever made to original nvme-cli code, to compile:

```
device-name:~/nvme-cli$ ninja –C .build
device-name:~/nvme-cli$ meson install –C .build
```

Test Script Architecture


1. invoke many times with different parameters
nvme-cli
instrumentation
test automation script
2. invoke after running nvme-cli
instrumentation log file
log analyzer script
Code path analysis result

I added instrumentation in nvme-cli code to generate log output for different branches. The instrumentation is saved to a log file. A log analyzer script then takes the log file and organizes the logs to display the different paths the specific commands took in nvme-cli. Then, to automate this process, I created an automation script that ran commands many times with different parameters and at the end ran the log analyzer script.

Instrumentation

I manually added the instrumentation code into nvme.c in json format at each branch. The instrumentation included details about the function it was in and the branches. Each line of instrumentation also included a unique id number to identify code paths. The instrumentation would be logged onto *logger.txt*.

Example instrumentation:

```
fprintf(fptr, "{\"type\": \"instrumentation\", \"id\": 54, \"func\": \"submit_io\", \"branch_name\": \"!(opcode & 1) && etc\",
\"state\": false, \"result\": \"success\", \"msg\": \"%s: Success\"}\n", command);
```

Each command hits multiple branches during the execution, so after one run of a command, multiple lines of such instrumentation are logged and is separated by a “start_signal” line of instrumentation.

Example of instrumentation log output from a single command execution:

```
{"type": "start_signal", "param": " read /dev/nvme0n1 -s 0 -c 0 -z 0 -d
readbuf" }
{"type": "instrumentation", "id": 4, "func": "read_cmd", "branch_name": "",
"state": null}
{"type": "instrumentation", "id": 5, "func": "submit_io", "branch_name": "",
"state": null}
{"type": "instrumentation", "id": 6, "func": "submit_io", "branch_name":
"opcod != nvme_cmd_write", "state": true}
{"type": "instrumentation", "id": 84, "func": "parse_and_open", "msg":
"RETURN VALUE OF argsconfig_parse: 0"}
{"type": "instrumentation", "id": 85, "func": "parse_and_open", "msg":
"RETURN VALUE OF get_dev: 4"}
{"type": "instrumentation", "id": 14, "func": "submit_io", "branch_name":
"!cfg.namespace_id", "state": true}
{"type": "instrumentation", "id": 22, "func": "submit_io", "branch_name":
"strlen(cfg.data)", "state": true}
{"type": "instrumentation", "id": 26, "func": "submit_io", "branch_name":
"!cfg.data_size", "state": true, "msg": "data size not provided"}
```

Log Analyzer

I created the log analyzer script *loggerReader.py* that would take the lines of instrumentation on *logger.txt* and create a String value listing the id numbers of the instrumentation logged from each command. The String value displays what paths each command went through, and all the commands were sorted out by grouping together commands that went through the same path. Output printed onto the Terminal by a command were also included under that command. This would be recorded on *log_dict.txt*.

Example:

```
----PATTERN----_3_5_7_14_22_28_39_46_51_54_56_SUCCESS
 write /dev/nvme0n1 -s 0 -c 0 -z 500 -d writebuf
Rounding data size to fit block count (512 bytes)
write: Success
 write /dev/nvme0n1 -s 0 -c 500 -z 500 -d writebuf
Rounding data size to fit block count (256512 bytes)
write: Success
----PATTERN----_3_5_7_14_22_28_39_46_49_56
```

```
 write /dev/nvme0n1 -s 0 -c 1500 -z 5000 -d writebuf
Rounding data size to fit block count (768512 bytes)
submit-io: Invalid argument
 write /dev/nvme0n1 -s 0 -c 1500 -z 5500 -d writebuf
Rounding data size to fit block count (768512 bytes)
submit-io: Invalid argument
```

Test Automation Script

The script *loggerRevised.py* invokes nvme-cli with commands multiple times through for loops, and parameters for commands are systematically changed each run. At the end, it runs the log analyzer script *loggerRead2.py*.

The following is a summary of what *loggerRevised.py* looks like:

```
def func_check():
      log = os.system("python3 ./loggerReader2.py")

// Set command (write) that will be run
cmd_format = "nvme write /dev/nvme0n1 -s {input1} -c {input2} -z 
{input3} -d readbuf"

// for loops systematically input arguments into commands
// range and intervals can be adjusted
for i in range(0, 1024 * 10, 500):
      for j in range(0, 1024 * 10, 500):
            for k in range(0, 1024 * 10, 500):

                         input1 = i
                         input2 = j
                         input3 = k

                         cmd = cmd_format.format(input1=input1,
      input2=input2, input3=input3)
                         result = os.system(cmd)
// Runs function func_check() which runs loggerReader2.py
new_count = func_check()
```

In future runs, if different commands want to be run, the value of cmd_format can be edited. To change the range and/or intervals at which the parameters change, edit the for loops. For loops can be added or taken off with the input variables to accommodate with more or less parameters in different commands.

Running Logger

(May need to install python3 first)

```
device-name:~/nvme-cli$ cd LOG/
device-name:~/nvme-cli/LOG$ sudo python3 loggerRevised.py
```

Code Location

All relevant files to this project and those mentioned in this report have been uploaded to https://github.com/dkminsoo/ssd-testing. Most files relating to logging are inside the *LOG* folder inside folder *nvme-cli*.

Logger Code and Run Results

Inside the folder *LOG* in nvme-cli, files and programs for logging are found. The main files are *loggerReader2.py*, *loggerRevised.py*, *log_dict.txt*, and *logger.txt*.

Results

The results from running commands are shown on the following table. The CMD column show which commands were run, the number and range of inputs, the interval at which the inputs were changed, and total number of commands run. The Symptoms column show what nvme-cli printed out in the terminal. The Frequency column shows the percentage of each symptom shown throughout all the times each command was run with varying inputs.
For a raw results, see *logger2.txt* in the GitHub repository.

CMD			**SYMPTOMS**				**FREQUENCY**
read			data size not provided				3.70%
(-s 0-10000, -c 0-10000, -z 0-10000)							
x500			Rounding data size to fit block count (bytes)				83%
9261			submit-io: Invalid argument				
			Rounding data size to fit block count (bytes)				10%
			read: Success				
			read: Success				3%
write			data size not provided				3%
(-s 0-10000, -c 0-10000, -z 0-10000)							
x500			write: Success				2.50%
9261							
			Rounding data size to fit block count (bytes)				9.50%
			write: Success				
			Rounding data size to fit block count (bytes)				85%
			submit-io: Invalid argument				
resv-acquire			RETURN VALUE OF argsconfig_parse:				9%
(-n 0 -c 0-1000 -p 0-1000 -i 0-1000 -a 0-1000 -t 0-1000)			RETURN VALUE OF get_dev:				
x500			NVMe status: Invalid Command Opcode:				
243			A reserved coded value or an unsupported value				
			in the command opcode field(0x2001)				
			Expected byte argument for '' but got ''!				91%
			RETURN VALUE OF argsconfig_parse:				
			Returning argsconfig_parse:				
resv-register			RETURN VALUE OF argsconfig_parse:				9%
(-n 0 -c 0-1000 -k 0-1000 -i 0-1000 -r 0-1000 -p 0-1000)			RETURN VALUE OF get_dev:				
x500			NVMe status: Invalid Command Opcode:				
243			A reserved coded value or an unsupported value				
			in the command opcode field(0x2001)				
			Expected byte argument for '' but got ''!				91%
			RETURN VALUE OF argsconfig_parse:				
			Returning argsconfig_parse:				
resv-release			RETURN VALUE OF argsconfig_parse:				3.60%
(-n 0-10000 -i 0-10000 -a 0-10000)			RETURN VALUE OF get_dev:				
x500			NVMe status: Invalid Command Opcode:				
9261			A reserved coded value or an unsupported value				
			in the command opcode field(0x2001)				
			Expected byte argument for '' but got ''!				96.40%
			RETURN VALUE OF argsconfig_parse:				
			Returning argsconfig_parse:				