SANDIA REPORT
SAND2020-8215
Printed August 2020

Sandia National Laboratories

Multi-Node Program Fuzzing on High Performance Computing Resources

Christian R. Cioce, Nasser Salim, Brian Rigdon, Danny Loffredo

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico
87185 and Livermore,
California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Rd
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods/

ABSTRACT

Significant effort is placed on tuning the internal parameters of fuzzers to explore the state space, measured as coverage, of binaries. In this work, we investigate the effects of the external environment on the resulting coverage after fuzzing two binaries with AFL for 24 hours. Parameters such as scaling to multiple nodes, node saturation, and parallel file system type on HPC resources are controlled in order to maximize coverage. It will be shown that employing a parallel file system such as IBM's General Parallel File System offers an advantage for fuzzing operations, since it contains enhancements for performance optimization. When combined with scaling to two and four nodes, while simultaneously restricting the number of coordinated AFL tasks per node on the low end (10-50% of available physical cores), coverage may be enhanced within a shorter period of time. Thus, controlling the external environment is a useful effort.

This page left blank

CONTENTS

LIST OF FIGURES

ACRONYMS AND DEFINITIONS

Abbreviation	Definition
AFL	American Fuzzy Lop
CGC	Cyber Grand Challenge
CLI	command line interface
CPU	central processing unit
DARPA	Defense Advanced Research Projects Agency
GPFS	General Parallel File System
HPC	high-performance computing
I/O	input/output
OS	operating system
PC	program counter
SRN	Sandia restricted network
VR	vulnerability research

This page left blank

1. INTRODUCTION

Fuzzing is an important piece of the modern vulnerability research (VR) methodology. According to an informal survey of vulnerability researchers, fuzzing is the most historically successful technique for finding bugs. [1] Fuzzers are an active area of security research because improvements can lead to the discovery of new vulnerabilities in software that matters. [2]

In this work, we make use of the American Fuzzy Lop (AFL) fuzzer to investigate two binaries with known bugs. We seek to understand the best method of running AFL on a high-performance computing system to achieve the most coverage during 24 hours of fuzzing. We introduce three ways to execute AFL, which differ in the way that tasks are bound to available cores. We show that performance (i.e. coverage) can be boosted by a variety of factors, including the number of cores per node utilized as well as the type of parallel file system employed.

Lastly, we introduce a tool for rapidly establishing a fuzzing campaign on high-performance computing (HPC) resources. The tool is named BlueClaw and it is a Python script designed with flexible input options. This tool is intended to serve VR researchers and analysts to orchestrate fuzzing at scale across an HPC system.

This page left blank

2. BACKROUND

2.1. American Fuzzy Lop

The most popular fuzzers today are based on an evolutionary code coverage metric. AFL is an evolutionary fuzzer written and maintained by Michal Zalewski. [3] AFL is one of the most popular fuzzers today and is credited with finding many previously undiscovered vulnerabilities. [2] AFL's primary mechanism for genetic mutation is based on code coverage.

To use AFL, the target binary must be recompiled with additional instrumentation to track code coverage. AFL will reserve an area of memory to store hit counts for each branch executed by the program. A hash of the **(source PC, destination PC)** is used as an index into the area, and a counter is incremented on every execution of each branch. Since a hash is utilized and collisions may occur, the code coverage is lossy, but the hash calculation is faster than tracking every branch precisely. This hit count hash table becomes the basis for the next phase of AFL, analyzing the execution for interesting behavior.

NOTE: Above, PC is an acronym for program counter. It is a reference to the address of the instruction pointer.

Once the instrumented program has executed with a given input, AFL needs to determine if the code coverage map contains any new paths. If so, the input is saved in a queue for future mutation. To make this determination, first, the hit counts are binned into a power-of-two bucket, so that minor differences in hit counts will be ignored. Then, the hit counts are compared against a cumulative code coverage map, checking for new paths or new hit counts. When the input file causes a crash or a timeout, the input is triaged against existing findings and saved in a separate directory.

AFL continues picking an input from the queue, mutating it, checking for new coverage, and potentially adding to the queue, until the user ends the fuzz session. AFL contains other heuristics; for example, new test cases are trimmed to the smallest size that maintains the same code coverage before they are mutated. For more details on the implementation of AFL, see the AFL README file. [4]

This page left blank

3. EXPERIMENTS AND METHODS

This work is a follow-on project from the single-node fuzzing work in 2019, [5] and extends the scope of fuzzing beyond a single node to two and four nodes. The two binaries under test in this work are "CROMU-9" [6] and "NRFIN-78" [7] from the 2016 Defense Advanced Research Projects Agency (DARPA) Cyber Grand Challenge [8] (CGC) set of binaries, which contain known bugs. These binaries were fuzzed with AFL version 2.52b [3] for 24 hours. We investigate the coverage as a function of node load (or node saturation) on a single node as well as two and four nodes. The fuzzers are always run in a coordinated fashion, and their output written to one of two types of shared parallel file systems: Lustre or a General Parallel File System (GPFS).

3.1. Sandia Restricted Network High Performance Computing Resources

In an effort to not distort any results from the original thrust, [5] we have performed all computations on the same resources at Sandia National Laboratories on the Sandia Restricted Network (SRN). The details for the platforms are thus repeated below for completeness.

The SRN HPC cluster Ghost consists of 26,640 compute cores, with each compute node having dual sockets with eighteen cores each (2 x 2.1 GHz Intel Broadwell® E5-2695 v4) and 128 GB of RAM. Note that the processors are hyperthreaded, each having eighteen cores and 36 threads for a total of 36 cores and 72 threads per node. Full hardware specifications can be found at Sandia's High Performance Computing website. [9]

Additionally, in this follow-on work we used two parallel file systems: Lustre and a GPFS. Details extracted from the Sandia HPC resource site are given here. "GPFS1 is our implementation of the IBM® General Parallel File System (GPFS™). It is a high performance shared-disk file management solution that provides fast, reliable network file system access to data from multiple nodes in a cluster environment. It provides storage on the SRN for pre and post process files and provide an alternative space to run jobs for users who may be having performance or other issues with the Lustre file systems." [10]

3.2. Single-Node and Multi-Node Fuzzing

There are many active efforts to improve coverage-guided fuzzing, most of which focus on enhancing the fuzzer itself. [11] [12] [13] [14] [15] [16] While this is unquestionably an extremely important research focus, our work seeks to tweak the environment in which fuzzers are used. We employ the standard off-the-shelf, or "stock", AFL fuzzer and run it under different conditions on a high-performance computing cluster to try to identify the most optimal conditions that lead to maximum coverage. Rather than tuning the internal parameters of the AFL fuzzer, we instead tune the external parameters. We believe this is a useful experiment as many security and vulnerability researchers will undoubtedly use readily available fuzzers in a standard fashion with default settings, rather than making changes to the internals and recompiling. It is therefore important to understand how the standard AFL fuzzer will behave under a variety of conditions in order to treat the output in a fair manner.

This page left blank

4. SINGLE-NODE FUZZING

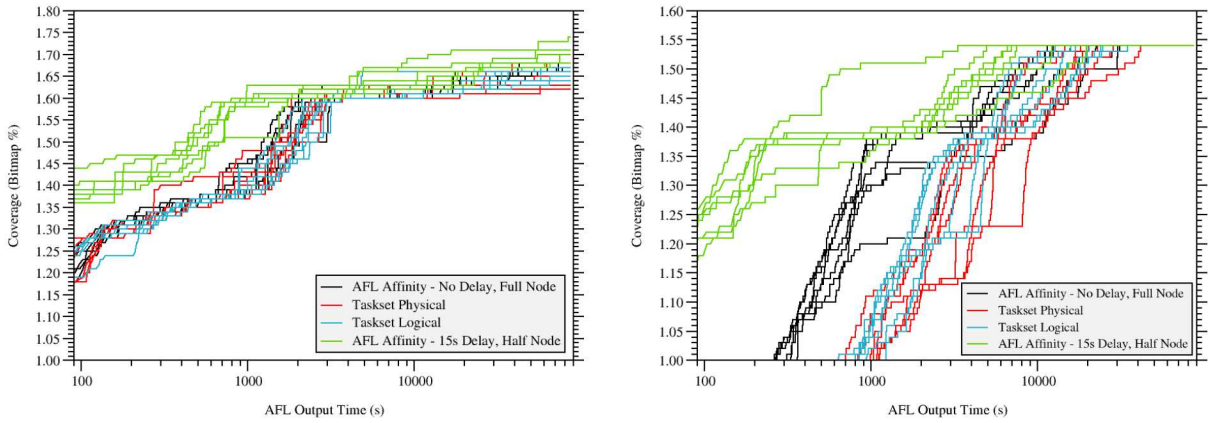
Our prior work [5] focused exclusively on single-node fuzzing, and we present a summary of the results here. The two CGC binaries under test in the current work, CROMU-9 and NRFIN-78, were also used as test cases in the prior work. For the single-node fuzzing study, only the Lustre network parallel file system was utilized.

4.1. AFL Types

We defined three methods for pinning AFL processes to cores on a single node: (1) AFL Affinity, (2) Taskset Physical, and (3) Taskset Logical. The AFL Affinity method simply allows AFL to assign fuzzers to any available cores. This is the ideal way to run AFL, however it was quickly discovered that unless sufficient time (we chose five to fifteen seconds) is provided between subsequent core assignments, a race condition occurs between AFL and the Linux OS whereby multiple processes are bound to a single core. In addition to the time intervention between subsequent task pinning, another solution is to manually pin tasks to available cores using the taskset command.

To ensure that AFL does not bind its tasks to cores, the *bind_to_free_cpu()* function call in afl-fuzz.c must first be commented out. One should then recompile AFL before executing it. There are two methods of using taskset; one requesting only physical cores, and another requesting a combination of physical and virtual cores, if available. Taskset Physical refers to only requesting physical cores for binding tasks, while Taskset Logical requests a pair of physical and virtual cores. Given our configuration on the SRN HPC resource Ghost, the nodes we are utilizing consist of 36 physical cores and 36 logical cores. For our specific hardware, cores #0-35 are physical and cores #36-71 are logical.

When comparing the resulting coverage as a function of time for the three methods above using a set of seven samples for each, the AFL Affinity method with a fifteen second time delay in addition to only saturating a node to half capacity was the clear winner (especially early in time). These results are presented in Figure 1. Note that there is no time delay introduced when manually pinning tasks to cores via taskset.



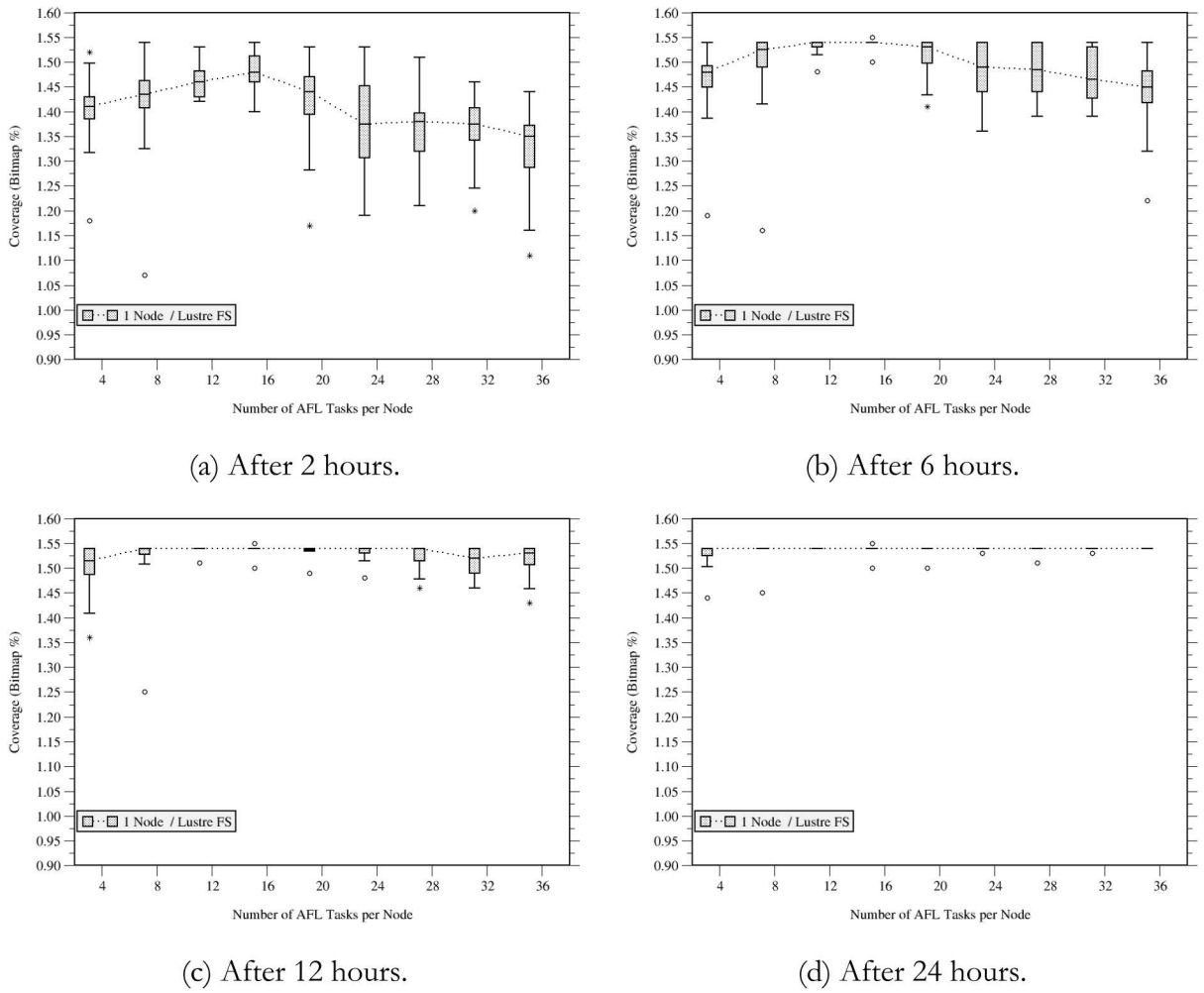
(a) CROMU-9 (b) NRFIN-78

Figure 1. Composite plot of the coverage as a function of AFL output time for AFL Affinity (black), Taskset Physical (red), Taskset Logical (blue), and AFL Affinity with a time delay (lime green) for the CROMU-9 and NRFIN-78 binaries.

4.2. Effect of Node Saturation

Restricted to the NRFIN-78 binary only, the effect of node load was investigated to determine if there is an optimal number of cores to activate for fuzzing on a single node. These results will be shown again in the Multi-Node Fuzzing section.

Running AFL via AFL Affinity mode with a fifteen second time delay between subsequent tasks and allowing each fuzzing instance to run in a coordinated fashion (i.e. share progress), we progressively saturate a single node with tasks and observe the trend of coverage as a function of node load. Each fuzzing campaign employed twenty samples and was allowed to fuzz the NRFIN-78 binary for 24 hours. The box and whisker plots below show the coverage at a particular time slice over the 24-hour period. These results are presented in Figure 2, but will be discussed in detail in the Multi-Node Fuzzing section.



(a) After 2 hours.

(b) After 6 hours.

(c) After 12 hours.

(d) After 24 hours.

Figure 2. Coverage as a function of node load (number of AFL tasks) over time for the NRFIN-78 binary with AFL Affinity plus a fifteen second time delay. Early in wall time, the median of the boxes trends upward until it peaks at sixteen cores (tasks). While no number of AFL tasks is statistically preferred at any time, the median point is highest at sixteen tasks for the first six to eight hours of fuzzing.

5. MULTI-NODE FUZZING

When scaling fuzzing campaigns to multiple nodes, we adopt similar principles to those utilized in the single-node case study, namely, using AFL Affinity mode with a time delay between concurrent instances (though we reduce the time to a five second wait period), and running tasks in a coordinated fashion so each fuzzing instance can share its progress. We investigate the performance of a fuzzing campaign by measuring the coverage over time as a function of node saturation. Recall that each node has 36 physical cores in our configuration. In these studies, we compare the coverage for campaigns utilizing one, two, and four nodes on two different parallel network file systems: Lustre and GPFS. As in our prior work each campaign consists of twenty samples per data point, for the purpose of constructing the box and whisker plots displayed below.

5.1. Lustre Parallel File System

The results presented in this subsection are those obtained from I/O operations on a network shared Lustre parallel file system. This file system is designed to be scalable and is widely implemented on clusters with HPC applications. According to the official website, "early adopters of Lustre were the Department of Energy National Laboratories including Lawrence Livermore, Sandia, Oak Ridge and, more recently, Los Alamos' Cielo supercomputer is supported by the Lustre file system." [17] This showcases the sustained interest and acceptance of the file system.

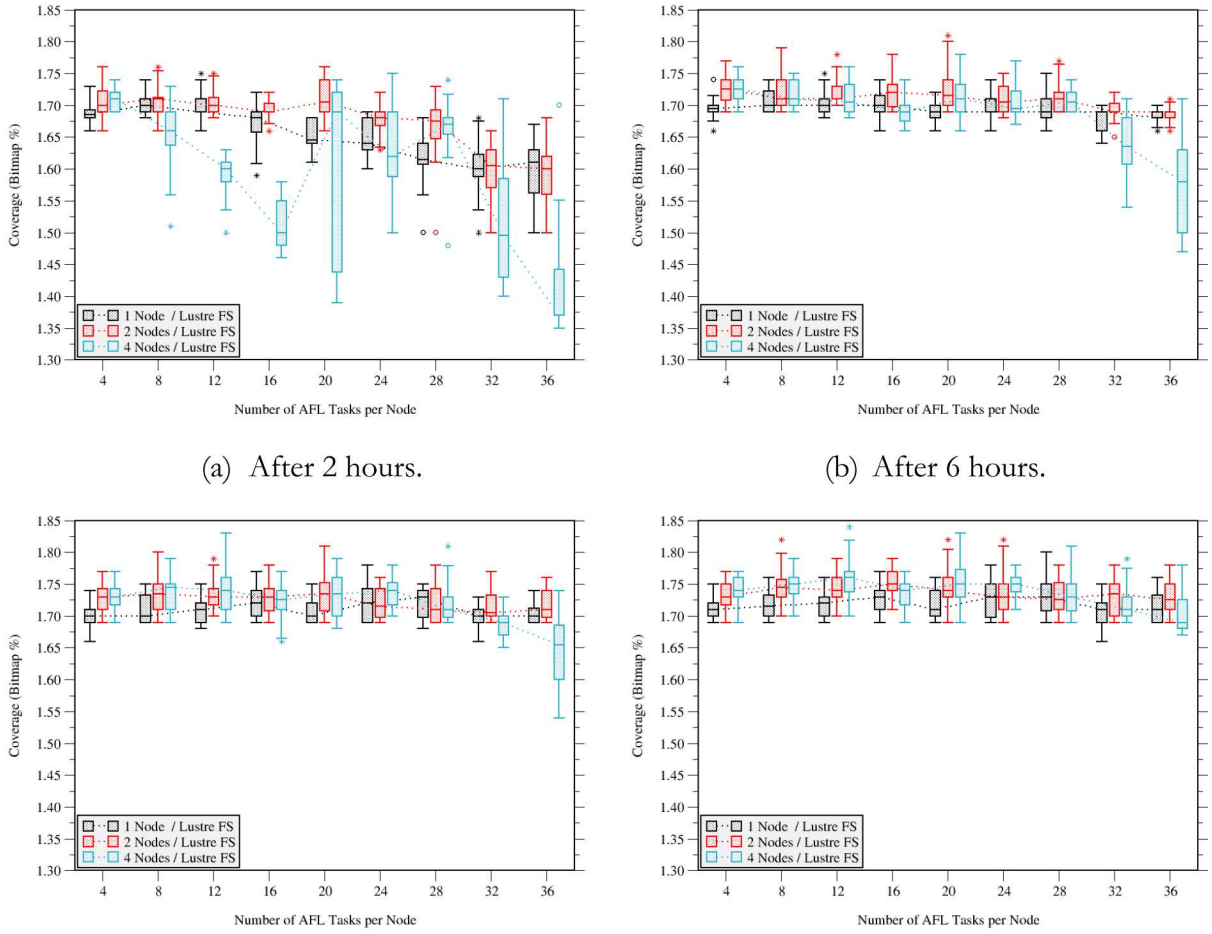
5.1.1. CROMU-9

The coverage plots in Figure 3 are presented on a node normalized basis. That is, the total number of AFL tasks are normalized to a single node which is 36 cores on our architecture. This allows for comparison of results across varying number of nodes.

NOTE: In an attempt to declutter the box and whisker plots, we present each x-axis value as boxes staggered side by side as opposed to overlapping one another. This can create the illusion that there are different x-values for neighboring black, red, and blue boxes, which is not the case. Each "cluster" is a set composed of one unique colored box and has the same x-value for each item in the set.

Very early in time, at the two-hour mark (Figure 3a), it is difficult to decipher the best node/task count combination that would result in the most coverage. The binary simply has not been under test long enough for AFL to make considerable progress. However, it does immediately seem that more progress can be made initially by selecting a lower task count per node.

After six hours, the two-node data (red shaded boxes) begins to outperform the one-node data for all but very high task counts. At the twelve-hour mark, the four-node data is in line with (and in some cases exceeds) the two-node data up to about 24 tasks per node. At the limit of 24 hours, the median value of the boxes for the four-node cases often exceeds that of the one- and two-node cases. While these data do not represent statistically unique values (that is, there is vertical overlap between neighboring boxes), it suggests that distributing the workload to multiple nodes can be an effective procedure for increasing coverage. In all cases, it is prudent to restrict the number of AFL tasks per node to some value less than approximately 75-80% of the available physical cores. Performance beyond this range is degrative.



(a) After 2 hours.

(b) After 6 hours.

(c) After 12 hours.

(d) After 24 hours.

Figure 3. Coverage plots for the CROMU-9 binary on a Lustre file system. Early in time, it is beneficial to distribute the workload from one to two nodes, so long as approximately 50% or less of the physical cores are utilized for coordinated fuzzing. Later in time, the benefit of scaling further to four nodes is realized.

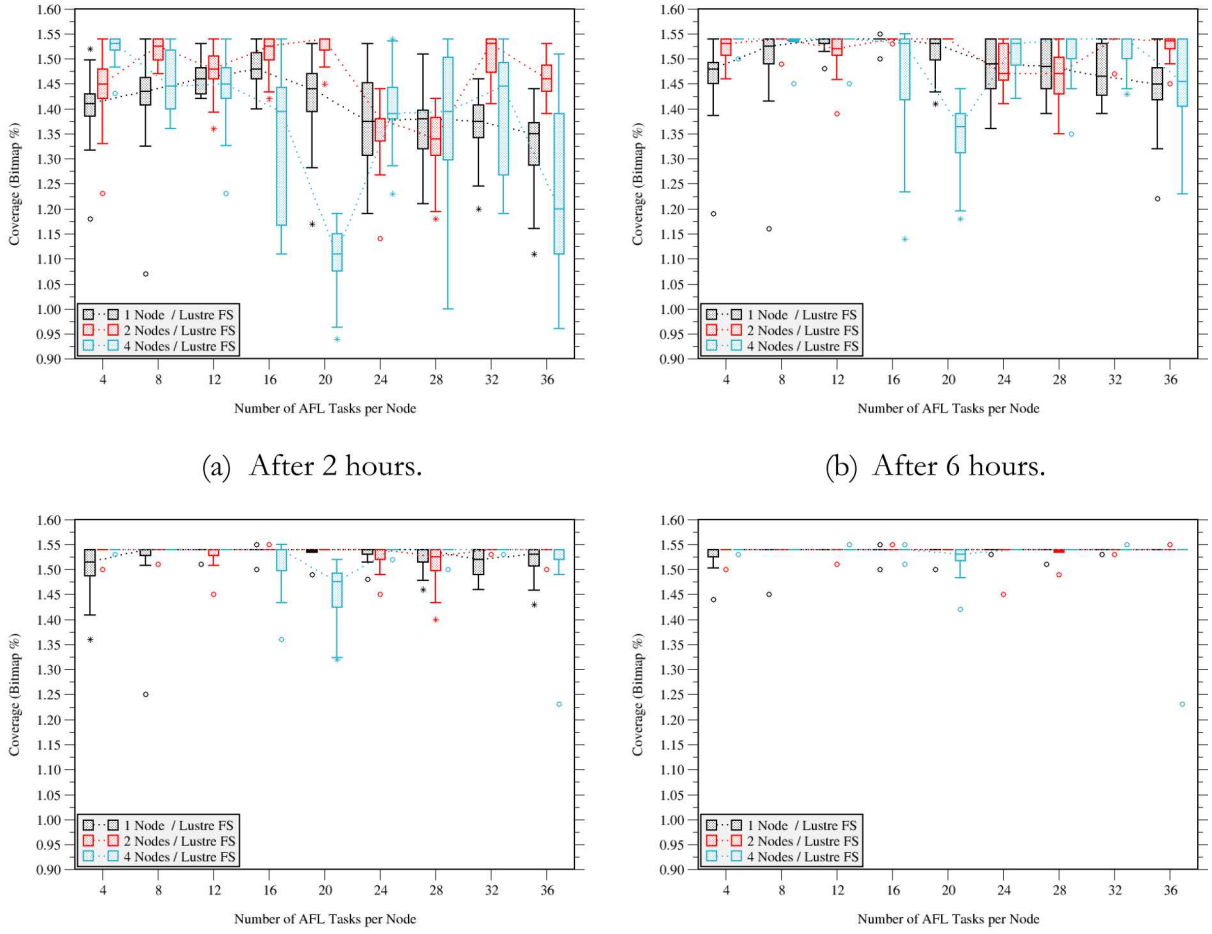
The trend of the four-node boxes for 32 and 36 AFL tasks per node is interesting and could potentially reach or exceed the median values of the one- and two-node data if allowed to continue beyond 24 hours.

5.1.2. NRFIN-78

For the NRFIN-78 binary we have prior results from the Single-Node Fuzzing work (see Figure 2, above). This data is repeated below in Figure 4 for consistency and is colored identically (black shaded boxes).

Similar to the CROMU-9 results after two hours, the boxes and whiskers tend to span a relatively large range for all node counts. In line with the generally accepted practices in the fuzzing community, it is recommended to fuzz for a longer period of time. After six hours of fuzzing, it appears that an upper bound of 1.54% is being reached by several data points. Keeping focus on the short end of the tasks per node scale (four to twelve), we start to see somewhat of a trend emerge in

favor of higher node count. It does not fully hold over this range, but it seemingly suggests that it is possible to increase coverage in a short time span by increasing the total number of AFL tasks while simultaneously distributing them across more nodes. Thus, taking the first (left-most) cluster into consideration in Figure 4b, the black box represents four total tasks across a single node, while the red and blue boxes represent eight total tasks across two nodes and sixteen total tasks across four nodes, respectively. Clearly, the variability among twenty samples diminishes as more tasks are included in a distributed fashion. Interestingly, this is also the case when activating all sixteen tasks isolated to a single node. While that holds true in this case, it does not hold in an absolute sense for all binaries under test and in all conditions.



(a) After 2 hours.

(b) After 6 hours.

(c) After 12 hours.

(d) After 24 hours.

Figure 4. Coverage plots for the NRFIN-78 binary on a Lustre file system. Early in time (two hours) scaling from one to two nodes is helpful, so long as approximately 50% or less of the physical cores are activated. However, scaling further to four nodes requires additional time to be beneficial.

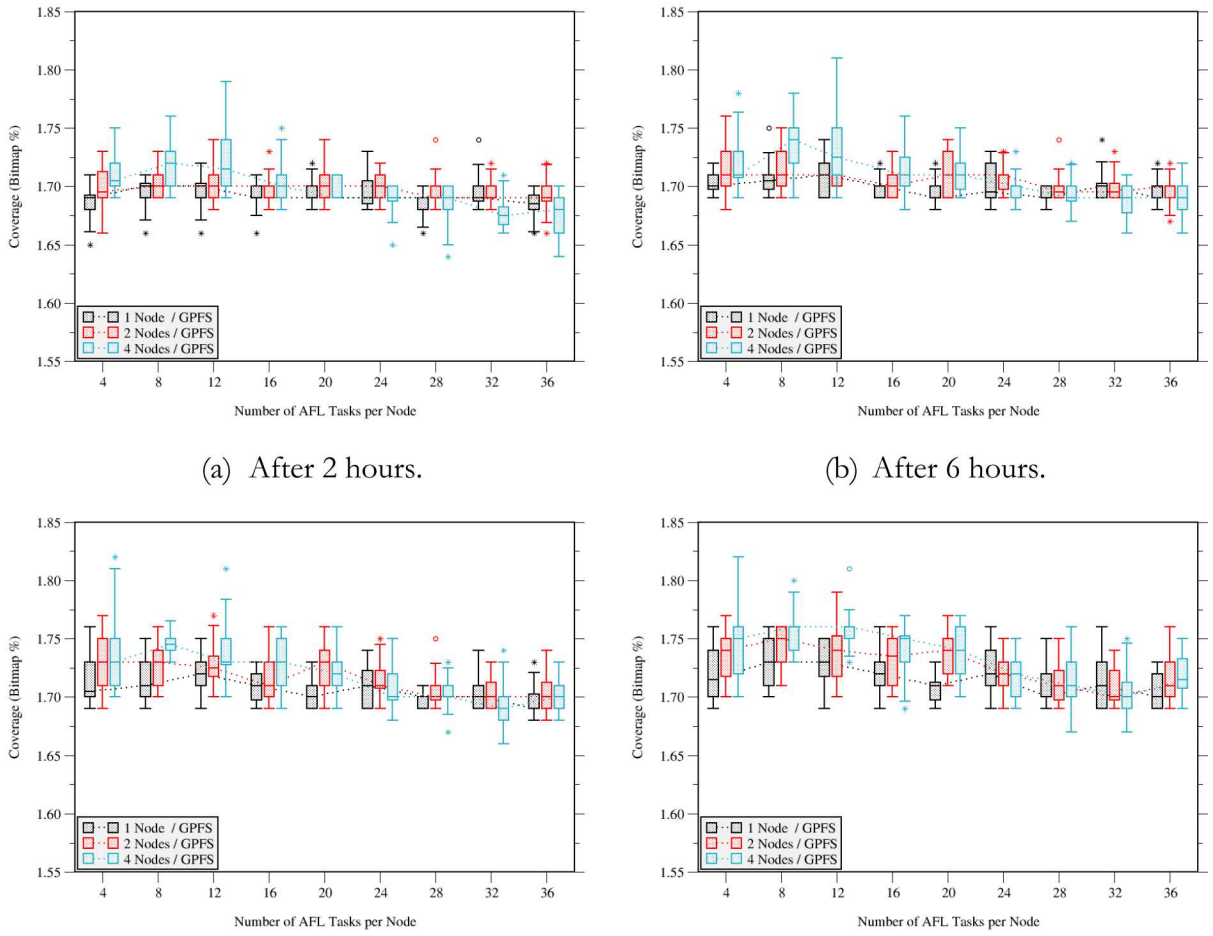
After twelve hours a few of the boxes have fully collapsed, and at the limit of 24 hours the majority of boxes have collapsed to the 1.54% value. We now turn our attention to comparing the above results with those using the GPFS file system. That is, we repeat the exact experiments but on a different type of parallel file system.

5.2. General Parallel File System

The results presented in this subsection are those obtained from I/O operations on a network shared General Parallel File System (GPFS). As noted above in the Sandia Restricted Network High Performance Computing Resources subsection, the GPFS-based scratch space at Sandia provides an alternative space to run jobs for users who may be having performance or other issues with the Lustre file systems. The GPFS has some inherent performance optimization features (from a caching and metadata viewpoint) [18] that may bode well for an application such as fuzzing, which performs substantial I/O on small files.

5.2.1. CROMU-9

The results depicted below in Figure 5 are comparable to those presented in Figure 3, which utilized the Lustre parallel file system. Note, however, that the scale of the y-axis in Figure 5 differs. This reduction in scope was made to increase data clarity.



(a) After 2 hours.

(b) After 6 hours.

(c) After 12 hours.

(d) After 24 hours.

Figure 5. Coverage plots for the CROMU-9 binary on a General Parallel File System. Note that the y-axis range is reduced, when compared with Figure 3 (Lustre parallel file system). The general trend of increased median value from one to two to four nodes is realized much sooner in time, as compared to the Lustre file system.

The most obvious observation occurs at the two-hour interval (Figure 5a), where the data is very tightly bound in bitmap percentage. This is a stark contrast to what is observed in Figure 3a. We also immediately observe the dominance of the median value for the four node dataset (blue boxes) up to sixteen tasks per node, a trend not realized until the twelve hour mark when the Lustre parallel file system is utilized (Figure 3c). The general shift upward of boxes and median values for the two-node and four-node datasets continues over time.

At the limit of 24 hours of fuzzing (Figure 5d), the data is similar to that in Figure 3d but with some higher box tops and whiskers. The lower bound of whiskers is similar. It is interesting to observe the trend of the right-most cluster at 36 tasks per node versus the Luster case. With GPFS, the trend of increased median values from one to two to four nodes is observed. This was postulated as possible in the case of the Lustre file system, if it had additional time to fuzz.

5.2.2. NRFIN-78

Similar to the observation above for CROMU-9, the reduction in variability of bitmap percentage for NRFIN-78 on GPFS versus Lustre after two hours also occurs (see Figure 6a). The scale of the y-axis in Figure 6 below is kept consistent with Figure 4 for a direct comparison between the two.

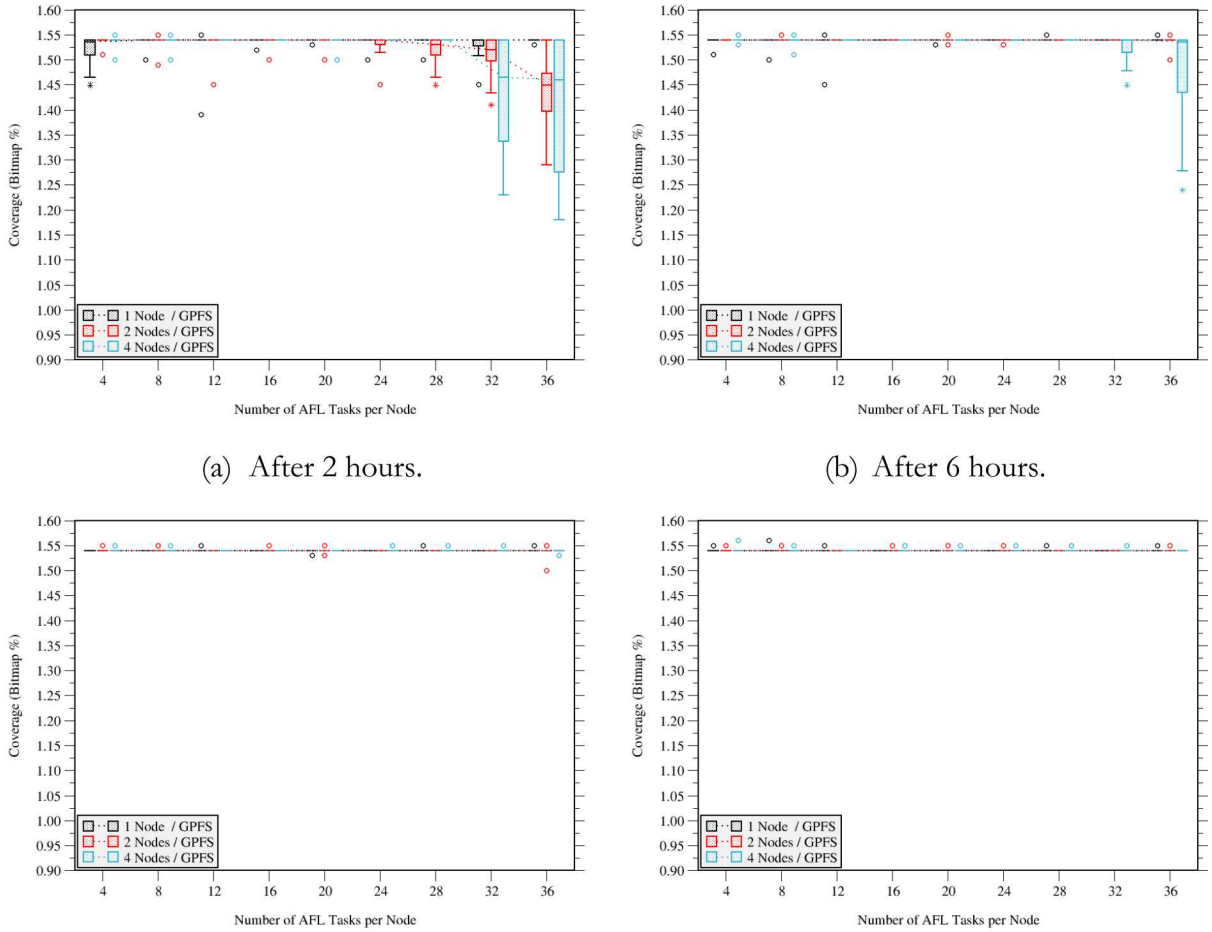
After only six hours of fuzzing (Figure 6b), and certainly after twelve hours (Figure 6c), the results are nearly representative of the data after a full 24 hours of fuzzing on the Lustre parallel file system (Figure 4d). In some cases, the coverage is increased within a fraction of the time as compared with the Lustre data. While it still remains a generally accepted best practice to fuzz binaries for no less than 24 hours, being able to tune external parameters and conditions to potentially boost coverage is powerful knowledge to have in the event that fuzzing under a deadline becomes necessary.

5.3. Best Practice Recommendations

For both the CROMU-9 and NRFIN-78 binary, greater coverage was achieved in less time when a fuzzing campaign was performed on a GPFS versus a Lustre parallel file system. The GPFS has more performance optimization features than Lustre, which could, in theory, lead to better fuzzing performance since fuzzers tend to perform a lot of I/O. Therefore, if fuzzing on an HPC, it is suggested to employ a GPFS. In general, however, it is wise to be mindful of the file systems available and to select one with the most performance optimization features.

In terms of node count and saturation, our studies reveal that fuzzing on more than 50% of available physical cores does not lead to enhanced coverage. In general, activating ~10-50% of physical cores (with one AFL task per core) is a best practice. Further, if more than one node is available, it may be advantageous to distribute the 50% over many nodes such that only ~10-25% of physical cores on each node is used. This practice can be considered wasteful from a CPU sharing perspective, as most of the cores on a node will be idle and off-limits to other users for the duration of the fuzzing session but may be necessary if fuzzing under a tight deadline.

Although we have observed these trends in the course of our experiments, actual performance may be affected by employing different hardware, network interconnects, system load, and even other binaries. There are many external parameters that can affect a fuzzing campaign, and we have done our best to remain consistent and generate reliable data.



(a) After 2 hours.

(b) After 6 hours.

(c) After 12 hours.

(d) After 24 hours.

Figure 6. Coverage plots for the NRFIN-78 binary on a General Parallel File System. Compared to the results in Figure 4 (Lustre parallel file system) the variability in coverage for almost all data points is significantly reduced, especially early in time.

6. BLUECLAW.PY TESTBED GENERATOR

BlueClaw is a tool aimed at rapid establishment of a fuzzing campaign at scale. It is written in Python and is designed for use on HPC systems. It is a command line interface (CLI) tool with several switches available for precise tailoring of a campaign. The original version was published in a prior SAND report [5], and has since been updated to include additional features and performance enhancements.

The tool is available on Sandia's Restricted Network. Please contact Sandia National Laboratories for access. Here, we present the README for the latest version, v1.2, which details usage and features available.

6.1. README.md

Automating the Workflow with BlueClaw

We wrote a Python script called BlueClaw which enables both HPC gurus and newbies alike to rapidly establish a testbed for fuzzing with AFL. The inclusion of command line options makes it very flexible and robust, tailoring the functionality to a user's wants and requirements. The initial release was Version 1.0, while the current release is Version 1.2 as of March 2020. Please contact Sandia National Laboratories for access.

Usage

usage: blueclaw.py [-h] [-A ACCOUNT] [-a AFL] [-e EMAIL] [-i INSTANCES] [-j JOBNAME] [-l] [-n INSTANCESPERNODE] [-p] [-s SAMPLES] [-t WALLCLOCK] [--version] **binDir mode**

positional arguments:

- **binDir** Specify the directory where the binary (or binaries) you seek to fuzz exist. Note that this specified path MUST be a relative path.
- **mode** Set the execution mode. 'SS' = Serial execution on a single binary, 'PS' = Parallel execution; single binary, 'SM' = Serial execution; multiple binaries

optional arguments:

- -h, --help show this help message and exit
- -A ACCOUNT, --account ACCOUNT Specify the WCID account number to use for HPC accounting.
- -a AFL, --afl AFL Specify the path to the afl-fuzz binary you wish to use for execution. Default is simply 'afl-fuzz', which assumes the default installed version of AFL is contained in one's $PATH. Specified path may be absolute or relative.
- -e EMAIL, --email EMAIL Provide your email address if you would like an email to be sent when the current job has completed.
- -i INSTANCES, --instances INSTANCES Specify the total number of AFL instances needed. This option is specific for parallel execution, and thus is used only in Mode PS.
- -j JOBNAME, --jobname JOBNAME Specify a unique name for a job. Can be helpful when many jobs are concurrent and paired with the --email option for bookkeeping purposes. Default is 'BlueClaw'.

- -l, --logical Set the flag to implement a combination of physical and logical cores (if available). Only valid for parallel fuzzing.
- -n INSTANCESPERNODE, --instances_per_node INSTANCESPERNODE Set the total (maximum) number of instances (active AFL coordinated fuzzers) for each node. This is only valid for multi-node campaigns, thus mode 'PS'. It is recommended to set this variable, so as to not overload a node with AFL fuzzers. Initial research recommends setting this as '>= NUM_PHYSICAL_CORES_PER_NODE / 2.0'.
- -p, --physical Set the flag to implement taskset physical mode, where AFL fuzzers are pinned to specific physical cores.
- -s SAMPLES, --samples SAMPLES Optionally specify the number of samples to collect for statistics purposes. While 1 is the default, our research made use of 20 samples per fuzzing campaign.
- -t WALLCLOCK, --time WALLCLOCK Set the time (wallclock limit) for the AFL job, in hours. 48 hours is the default and is also the max time allowed for most clusters. 1 hour is the minimum allowed.
- --version show program's version number and exit

Overview & Setup

The standard method of running BlueClaw is:

blueclaw.py <OPTIONS> BinDir Mode

The full list of available options is detailed with the --help option. BinDir is the path to the directory where the binary you wish to fuzz is located, and Mode is the desired mode to run the script in. There are currently three supported modes: SS (**S**erial execution against a **S**ingle binary), PS (**P**arallel execution against a **S**ingle binary), and SM (**S**erial execution against **M**ultiple binaries). Note that the most functionality will come from the PS mode, where development has been focused to support scalable fuzzing across multiple nodes. Also note that mode SM is largely experimental at this time.

The suggested process for creating a brand-new fuzzing campaign with BlueClaw is as follows:

1. Create a directory where all fuzzing jobs will run. Descend into it.
2. Copy the binary under test here. The current version of BlueClaw requires that you have a local copy of the binary to be fuzzed in the current directory.
3. Create a directory called “input” and fill it with any initial seeds specific for the binary under investigation. It is necessary to label this directory “input”.
4. Up to this point it is assumed that no prior fuzzing campaign has been run. If that is the case, you may now run BlueClaw. If, however, you wish to continue a previous campaign using a particular output as input, then simply create a directory called “output” and place the previously generated output in here.

Once this setup is complete, you are now ready to run BlueClaw. We recommend that the tool itself (blueclaw.py) exist outside of the directory where the fuzzing campaign will run. In the examples that follow, it will reside in a user's home directory.

Samples

BlueClaw is designed to generate N samples each time the tool is run. This number defaults to one (1), but can be specified with the -s flag, as:

```
~/blueclaw.py [OPTIONS] -s 20 …
```

Any positive integer from 1 to 100 is allowed. Our published research has made use of 20 samples per campaign in order to generate meaningful statistics.

When BlueClaw is run successfully, *N* "sample_XX" directories will be created in the current directory where the tool was executed. Even if only one sample is requested, the resulting "sample_00" folder will be created and populated with all of the necessary files needed for AFL to begin fuzzing. Note: The numerical identifier for samples is zero-based.

Syncing

Syncing of output among coordinated fuzzers within a campaign is currently unsupported in BlueClaw. To be clear, coordinated fuzzers are inherently in sync when the output is written to a networked file system (as was the case in our published research). However, this is not an ideal practice due to potential lagging and I/O bottlenecks. Therefore, we intend to build support for corpus syncing in subsequent versions.

Runtime

BlueClaw supports programmatically setting the total time you wish to run AFL against your binary under investigation. This is achieved with the presence of the -t flag. One point to note here is that one extra hour will be added to the requested HPC walltime to account for overhead and the presence of embedded sleep commands in the resulting *submit.sh* file that BlueClaw generates. So, if you are limited by a maximum 48-hour walltime request on the HPC system, your upper limit request with the -t flag should therefore be 47. BlueClaw will accept any integer, but the HPC scheduler will inevitably deny your request if you exceed system limits.

Cores Per Node

BlueClaw was built on, and designed exclusively for, execution on Sandia National Laboratories' Sandia Restricted Network (SRN, unclassified) HPC systems. It therefore has built-in logic and hardcoded parameters for the number of cores per node for particular resources. To extend the support for additional systems outside this range, you must edit the GetCoresPerNode() function of the script.

AFL Path

BlueClaw assumes that the "afl-fuzz" command is in one's path. Should it not be, or you desire to use a specific version of AFL, this can be achieved through the -a flag, as:

```
blueclaw.py [OPTIONS] -a ../afl-2.52b-comment/afl-fuzz …
```

> NOTE: The path to AFL, as specified by the -a option, may be either relative or absolute, but the path to the directory where the target binary/binaries under test are located must be a relative path.

Restarts

BlueClaw supports restarting prior fuzzing campaigns via an auto-detect mechanism. That is, if there is a directory labeled "output" in the current directory where BlueClaw is executed, and it is populated with data of any kind, BlueClaw will treat that data as output from a prior AFL session

and use it as input to the upcoming campaign. It will be copied redundantly into all *N* "sample_XX" directories as well.

There is currently no override switch for restarting a prior fuzzing campaign, so it is the user's responsibility to ensure that no directory labeled "output" exists in the directory where BlueClaw is run.

Examples

The following examples demonstrate how to run BlueClaw in PS and AFL Affinity mode, where AFL is allowed to pin tasks to cores using its own logic. BlueClaw does support manually pinning tasks to cores with Taskset Physical and Taskset Logical modes, but we do not go into detail here. Please see the published SAND paper (2019) for further details.

Specifying Total Instances for a Single Node

To run coordinated fuzzers (which we refer to as "parallel" – it is parallel in the sense of more than a single fuzzing instance sharing a corpus and focused on a common binary) on a single node, make sure that your instances request (-i flag) does not exceed the total number of physical cores available on your compute node. Instances are the total number of fuzzers that will be active against a common binary under investigation. If you are not careful, you can accidentally request more instances than available number of cores on a node. We will demonstrate this shortly. However, in this example we request 16 coordinated AFL instances (and thus 16 cores, as no more than one fuzzing instance will be bound to a single core) on a single node for 24 hours against a binary that we have a local copy of in the current directory.

```
~/blueclaw.py -t 24 -i 16 -j BlueClaw ./ PS
```

The -j option allows you to specify the job's name. It is "BlueClaw" by default, but we specify it here to showcase the feature. Note that the BinDir is ./. It does not require a trailing forward slash, and thus can simply be . if desired.

Specifying Instances per Node for a Multi-Node Campaign

Running BlueClaw with the intention of fuzzing across multiple nodes is similar to the above example. What you should specify are the total number of instances as well as the number of instances per node. Our research indicates that completely saturating a node with fuzzing instances is not ideal for maximizing coverage within a limited time frame, and so we offer the option to specify how many cores per node should be activated.

```
~/blueclaw.py -t 12 -i 32 -n 16 . PS
```

The above command requests 32 total coordinated AFL instances while only activating 16 cores per node. It will only run for 12 hours, as specified with the -t switch. As above, if we assume our compute nodes have 36 cores per node, then two nodes will be requested from the HPC scheduler and each node will have slightly less than half of its physical cores activated. Our preliminary research shows that this is near the optimal range of cores to activate for the best fuzzing performance, especially if only fuzzing for short periods of time (less than 12-16 hours).

Specifying Total Instances in Excess of a Single Node

In the current version of BlueClaw, if you request a total number of instances that happens to exceed the available number of cores per node but do not concurrently specify the number of instances per node desired, then the default behavior is to calculate the instances per node via

"NUM_TOTAL_INSTANCES / NUM_NODES". As an example, if you have compute nodes with 36 cores per node, and issue the following command:

```
~/blueclaw.py -i 46 . PS
```

then BlueClaw will understand that 10 fuzzing instances must spill over onto a second node (46 total instances minus 36 cores per node), and thus request 2 nodes. Again, the default behavior at this time is to divide the workload among each node by the total number of nodes needed for a particular job. In this instance, there would be two total nodes with 23 (46 / 2) cores activated on each node. We recommend specifying the desired instances per node, but nonetheless have attempted to build in a failsafe.

6.2. Suggested Usage

There are various modes and switches available in BlueClaw to tailor it for a particular desired setup. This flexibility makes it a powerful tool for rapidly establishing, or even continuing, a fuzzing campaign. The suggested process for generating a brand-new fuzzing campaign is briefly outlined in the README, but we will expand on it here.

6.2.1. Pre-Execution

As noted previously, BlueClaw is designed to be used in an HPC environment. More specifically, it is custom built for Sandia's production environment. There will be some slight modifications required before it will function properly, most notably in the GetCoresPerNode() function, where you must specify the hardware environment for the platform it will be run on. This is necessary for the resulting job submission script to contain accurate information, which will ultimately be processed by the scheduler.

6.2.2. Execution

The procedural steps for initiating a fuzzing campaign are listed in the README. It is recommended to create a new, empty directory for each fuzzing campaign.

```
$ mkdir Campaign_1
```

It is then necessary to copy the binary under test along with any input files (initial seeds) into this directory.

```
$ cd Campaign_1 && cp ~/bin/bin_under_test.afl.out .
```

```
$ cp -r ~/bin/input_files ./input
```

The initial seeds should be in a directory titled "input". This is a vital step, as BlueClaw looks for a directory with that name.

If this campaign happens to be a restart of a prior campaign, copy the previously generated output files into this directory in a subdirectory titled "output". Otherwise, BlueClaw can now be executed. There are three different examples of how to execute BlueClaw in the README above, and it is worth emphasizing that the most power will come from the PS mode, where a single binary under test is run in parallel (which is another way of saying there will be multiple fuzzers running in parallel). The latest version of BlueClaw supports coordinated fuzzing at scale via the PS mode.

After BlueClaw executes successfully, there will be a number of "sample_XX" directories created in the current directory, depending on how many samples were requested with the `-s` switch. Each one

of these "sample_XX" subdirectories is a fully contained fuzzing space. In order to initiate the campaign, simple descend into each sample subdirectory and issue the sbatch command on the job submission file. For example, assuming twenty samples were requested:

```
$ for i in {00..19}; do cd sample_${i} && sbatch submit.sh && cd ..; done
```

6.2.3. Post-Execution

The jobs submitted to the scheduler will timeout on their own. The length of time they will run for is determined by the -t switch. If no value is specified, they will default to run for 48 hours. If an email is provided during execution, a message will be sent to your inbox from the HPC system when each job has completed. To check in on a currently running job, the squeue command can be issued to see if a job has already started running or is pending to get started.

7. CONCLUSION

In this follow-on work we expand our fuzzing campaigns on HPC resources from a single node [5] to multiple nodes, up to four nodes. Two CGC binaries, CROMU-9 and NRFIN-78, were selected for investigation and fuzzed with AFL version 2.52b for 24 hours on two different parallel file systems: Lustre and GPFS. The hardware employed consists of 36 physicals cores per node and resides on the SRN HPC cluster Ghost.

Each fuzzing campaign consists of a set of 20 samples, in order to generate statistical box and whisker plots. We progressively saturate nodes with coordinated AFL tasks and measure the resulting coverage (bitmap percentage) over the course of 24 hours. Varying the degree of node load along with parallel file system type, we seek to understand how to best tune the environmental variables so as to maximize the coverage for a given binary under test.

It is a generally accepted practice to fuzz a binary for no less than 24 hours, however we have identified ways to accelerate coverage should time be a constraint. While the results may differ for a particular binary under test, we have observed increased coverage in a shorter time period for both the CROMU-9 and NRFIN-78 binaries when conducting fuzzing operations on a GPFS parallel file system as compared to Lustre. Lustre is a widely used and robust parallel file system designed for HPC applications, however GPFS contains additional performance enhancement features which are advantageous for applications performing substantial I/O operations, such as fuzzing.

Establishing a testbed for a fuzzing campaign, which in this case consisted of 20 samples per campaign, is a time-consuming task within itself. For our work, one campaign is equal to one box and whisker data point on the coverage plots presented above in the Single-Node Fuzzing and Multi-Node Fuzzing sections. Given two binaries under test, three node counts (one, two, and four nodes), and nine unique numbers of AFL tasks per node, we created 54 campaigns (each composed of 20 samples, for a grand total of 1,080 fuzzing experiments) in the scope of this work. In order to simplify the process and reduce the time required to build a campaign, we created a Python tool to automate the process. The resulting tool is called BlueClaw, and it is designed to operate exclusively on HPC systems. That is, it contains logic to seamlessly scale a fuzzing campaign to as many nodes as desired. It is a command line interface (CLI) driven application that offers significant flexibility and tuning of parameters to suit almost any need. To that end, it supports generating testbeds from scratch as well as continuing previously run fuzzing experiments. The tool is available internally on the SRN network at Sandia. Please contact Sandia National Laboratories for access.

This page left blank

REFERENCES

[1] M. Zalewski, "Understanding the process of finding serious vulnerabilities," 08 2018. [Online]. Available: https://lcamtuf.blogspot.com/2015/08/understanding-process-of-finding.html. [Accessed 02 05 2018].

[2] M. Zalewski, "American Fuzzy Lop (2.52b)," 2017. [Online]. Available: http://lcamtuf.coredump.cx/afl/#bugs. [Accessed 02 05 2018].

[3] M. Zalewski, "American Fuzzy Lop (2.52b)," 2017. [Online]. Available: http://lcamtuf.coredump.cx/afl/. [Accessed 02 05 2018].

[4] M. Zalewski, "README," 2017. [Online]. Available: http://lcamtuf.coredump.cx/afl/README.txt. [Accessed 02 05 2018].

[5] C. R. Cioce, D. G. Loffredo and N. J. Salim, "Program Fuzzing on High Performance Computing Resources," no. doi:10.2172/1492735, 2019.

[6] J. Rogers, "CROMU_00009: RAM-based filesystem," 1 February 2017. [Online]. Available: https://github.com/CyberGrandChallenge/samples/tree/master/cqe-challenges/CROMU_00009. [Accessed March 2020].

[7] M. Koo, "NRFIN_00078," 1 February 2017. [Online]. Available: https://github.com/CyberGrandChallenge/samples/tree/master/examples/NRFIN_00078. [Accessed March 2020].

[8] D. Fraze, "Cyber Grand Challenge (CGC)," DARPA, [Online]. Available: https://www.darpa.mil/program/cyber-grand-challenge. [Accessed 02 05 2018].

[9] Sandia National Laboratories, "Sandia National Labs High-Performance Computing," 2018. [Online]. Available: https://onestop.sandia.gov/hpc/178-cts-like-systems/242-ghost. [Accessed 30 04 2018].

[10] Sandia National Laboratories, "Sandia National Labs High-Performance Computing," 2019. [Online]. Available: https://onestop.sandia.gov/hardware/127-filesystem/73-gpfs1. [Accessed 2 12 2019].

[11] M. Zalewski, "The AFL++ fuzzing framework," [Online]. Available: https://aflplus.plus/. [Accessed 28 Apr 2020].

[12] "Honggfuzz," [Online]. Available: https://honggfuzz.dev/. [Accessed 28 Apr 2020].

[13] "libFuzzer," [Online]. Available: https://llvm.org/docs/LibFuzzer.html. [Accessed 28 Apr 2020].

[14] "FairFuzz," [Online]. Available: https://github.com/carolemieux/afl-rb. [Accessed 28 Apr 2020].

[15] "Driller," [Online]. Available: https://github.com/shellphish/driller. [Accessed 28 Apr 2020].

[16] "Angora," [Online]. Available: https://github.com/AngoraFuzzer/Angora. [Accessed 28 Apr 2020].

[17] Lustre, "About the Lustre File System," [Online]. Available: http://lustre.org/about/. [Accessed 14 Apr 2020].

[18] V. Dubeyko, "Comparative Analysis of Distributed and Parallel File Systems' Internal Techniques," 25 Apr 2019. [Online]. Available: http://arxiv.org/abs/1904.03997. [Accessed Apr 2020].

DISTRIBUTION

Email—Internal

Name	Org.	Sandia Email Address
Nasser Salim	05631	njsalim@sandia.gov
Christian R. Cioce	05636	crcioce@sandia.gov
Danny Loffredo	05638	dloffre@sandia.gov
Brian Rigdon	05684	jbrigdo@sandia.gov
Technical Library	01977	sanddocs@sandia.gov

This page left blank

Sandia National Laboratories

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.