SANDIA REPORT
SAND2019-0674
Printed January 2019

Sandia National Laboratories

Program Fuzzing on High Performance Computing Resources

Christian R. Cioce, Danny Loffredo, Nasser Salim

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

American Fuzzy Lop (AFL) is an evolutionary fuzzer that is strategically implemented as a tool for discovering bugs in software during vulnerability research. This work seeks to understand how to best implement AFL on the High-Performance Computing resources available on the unclassified network at Sandia National Laboratories. We investigate various methods of executing AFL, requesting varying numbers of tasks on single compute nodes with 36 physical cores and 72 total threads. A Python script called BlueClaw is presented as an automated testbed generator tool to assist in the tedious process of creating and executing experiments of any scale and duration.

This page left blank

CONTENTS

LIST OF FIGURES

ACRONYMS AND DEFINITIONS

Abbreviation	Definition
AFL	American Fuzzy Lop
CGC	Cyber Grand Challenge
CPU	Central processing unit
HPC	High-performance computing
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 (restricted to single nodes) to achieve the most coverage after 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 can be boosted in the first six to eight hours if only ~50% of the physical cores on a given node are used.

This page left blank

2. BACKGROUND

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

From the Cyber Grand Challenge [5] (CGC) suite, we selected the CROMU_00009 and NRFIN_00078 (hereinafter "CROMU-9" and "NRFIN-78") binaries for analysis. These two binaries were selected after we ran an initial test of all the CGC binaries for 24 hours. Both binaries exhibited the most coverage as compared to the others in the full set. The benefit of using the CGC binaries is that they were designed to represent real world bugs at varying difficulty levels, yet we have ground truth for each of the vulnerabilities in the corpus.

Each experiment was conducted on its own reserved node on the Sandia Restricted Network (SRN) High-Performance Computing (HPC) cluster Ghost, and was run for 24 hours. Further, each test was conducted seven times (unless otherwise noted) to collect statistics and we refer to each fuzz run as a "trial."

Both CROMU-9 and NRFIN-78 were subjected to the same fuzzing conditions: in both serial and in parallel (also referred to as uncoordinated and coordinated, respectively), and with three different "types," as detailed in the subsections below. The starting seeds used for both binaries were those that were provided by the CGC suite as valid inputs.

Our main interest is the measurable metric of code coverage. We seek to understand which method of running AFL will result in the most coverage in a given wall time, with repeatability. Early in our investigations, we observed a race condition occurring when running AFL with default settings. It was discovered that AFL's method of binding tasks to available cores was competing with that of the Linux operating system and thus there were multiple processes bound to a single core (see Figure 4-1 and Figure 4-2). For a set of seven trials, the race condition occurred at varying frequencies for each binary, but was always a majority occurrence. This sporadic nonuniform load balancing was unacceptable, so we manually set CPU affinities via the Linux command-line tool "taskset." [6] We will discuss taskset and its usage in Section Taskset Physical.

The remainder of the paper is organized as follows. First, we provide an overview of the hardware and computing environment used for our experiments, then we provide a description of our types of AFL configurations. The results for the various experiments are then presented and discussed in detail in Results. We also investigate the effect of saturating a node with AFL instances that run in both a coordinated and uncoordinated fashion. A brief comparison is made for fuzzing on a desktop machine versus an HPC node. Lastly, we introduce a tool called BlueClaw to automate the setup of these experiments so the reader can extend testing to any number of binaries, on any number of cores, for any number of hours.

3.1. Sandia Restricted Network High Performance Computing Resources

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. [7]

3.2. AFL Types

For this study, we have used AFL version 2.52b [3] exclusively. We define three different methods of executing AFL, referred to herein as "AFL Affinity," "Taskset Physical," and "Taskset Logical." We outline each in the subsections that follow, and then we present data in the Results section.

3.2.1. AFL Affinity

AFL Affinity refers to running AFL with default settings right out of the box. Specifically, we allow AFL to bind itself to any available cores on the machine where it is running by the prescribed method contained within the afl-fuzz.c source file that is distributed with AFL when downloaded.

3.2.2. Taskset Physical

For Taskset Physical, we first comment out the function call to *bind_to_free_cpu()* in afl-fuzz.c and then recompile. Additionally, execution of AFL is prepended by the taskset command. With taskset, we can set the CPU affinity manually, allowing us to bind a specific AFL instance to a particular core. The addition of the --cpu-list option to taskset allows us to specify the exact physical core to which each AFL instance should be bound. The syntax for taskset is as follows:

```
taskset --cpu-list <PHYSICAL_CORE_NUMBER> …
```

where the PHYSICAL_CORE_NUMBER is any single non-negative integer in the closed interval [0,35]. Note that our hardware contains 36 cores, thus the closed interval ends at 35. In general, the upper bound is MAX_CPU_COUNT - 1.

3.2.3. Taskset Logical

Like Taskset Physical, we first require that afl-fuzz.c be altered in the same manner as above and subsequently recompiled. However, what differs in Taskset Logical is the inclusion of a combined physical plus logical core to the --cpu-list parameter option of taskset, given as a pair. Recall from the hardware specifics above that the nodes we are utilizing consist of 36 physical cores and 36 logical cores. For our hardware, cores #0-35 are physical and cores #36-71 are logical. Thus, our taskset command line specification looks like:

```
taskset --cpu-list <PHYSICAL_CORE_NUMBER>,<LOGICAL_CORE_NUMBER> …
```

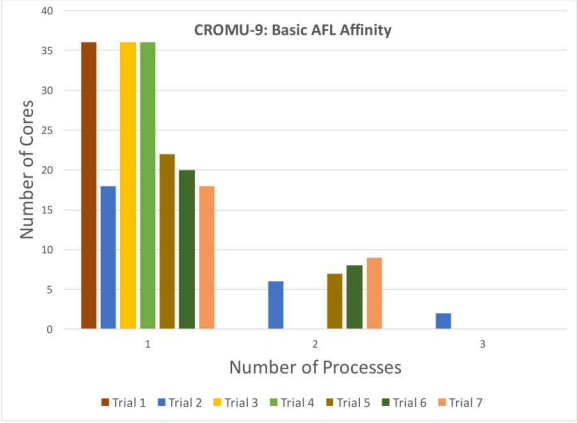
where the PHYSICAL_CORE_NUMBER is the same as above, and the LOGICAL_CORE_NUMBER is any single integer in the closed interval [36,71].

4. RESULTS

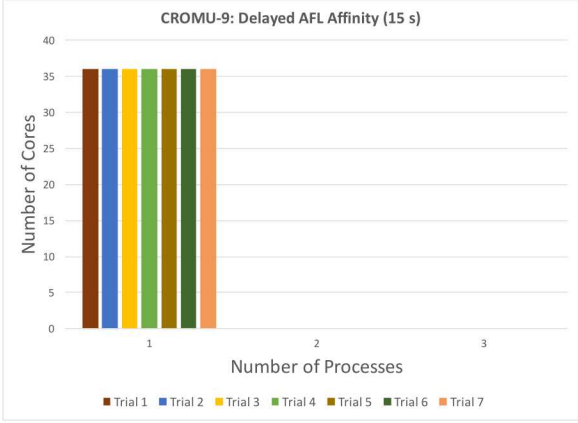
4.1. AFL Affinity

We fuzzed the CROMU-9 and NRFIN-78 binary using AFL via type AFL Affinity for 24 hours, and investigated the occurrence of over-binding processes to cores. A plot detailing this race condition for each trial is presented in Figure 4-1(a)&(c), and Figure 4-2(a)&(c).

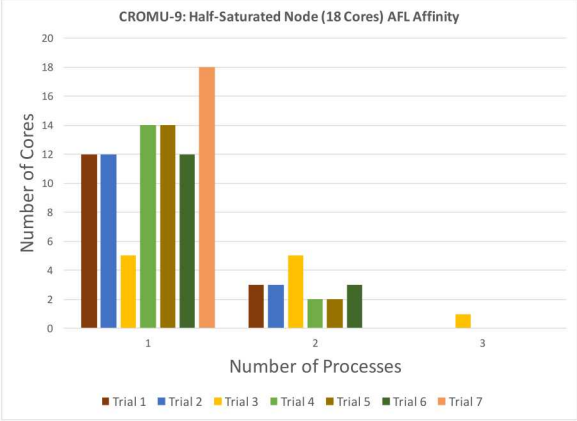
For the CROMU-9 binary (Figure 4-1(a)), trials one, three, and four did not experience a race condition: all 36 fuzzing instances were evenly distributed among available cores. However, trials two, five, six, and seven had more than one process bound to available cores. Trial two (blue bars) experienced the most extreme condition, with two cores having three processes bound and six cores having two processes bound.



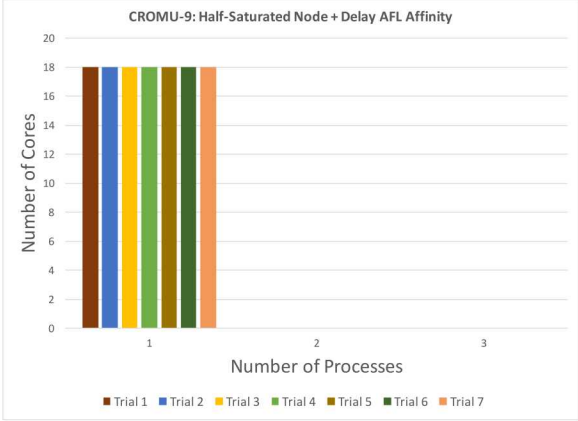
(a) Basic AFL Affinity



(b) Delay



(c) Half-Saturated Node

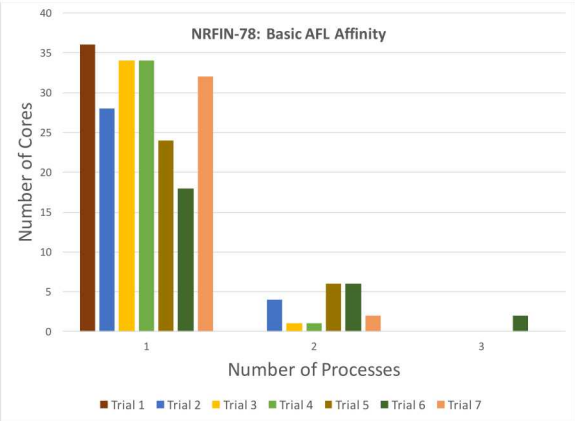


(d) Half-Saturated Node + Delay

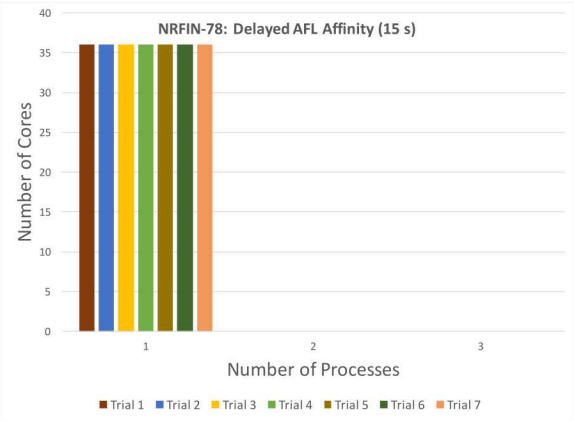
Figure 4-1. Plots of the number of fuzzing instances (i.e. processes or tasks) bound to available cores for CROMU-9 run with AFL Affinity. Using the default AFL settings (Figure 4-1(a)&(c)) leads to processor contention. The inclusion of a time delay between concurrent AFL execution calls fully resolves the issue, as can be seen in Figure 4-1(b)&(d).

For the NRFIN-78 binary (Figure 4-2(a)), only one trial (Trial 1) had all 36 processes distributed to unique cores. That is, 85% of the set (six out of seven trials) experienced a race condition. Trial six (dark green bars) experienced the most extreme case, with two cores having three processes bound and six cores having two processes bound.

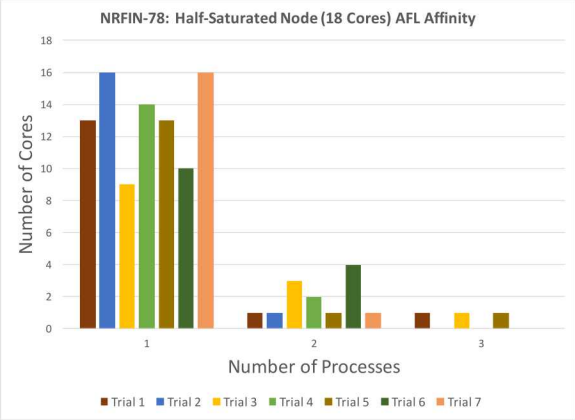
This processor contention issue can be resolved by a few different methods. One can insert a time delay between concurrent AFL calls to allow the Linux OS enough time to respond to AFL's assignment of tasks. We chose to wait fifteen seconds between execution calls, which solved the problem and led to the correct assignment of a single task per physical core, as can be seen in Figure 4-1(b)&(d), and Figure 4-2(b)&(d). An alternative solution is to manually assign tasks to cores without using AFL's *bind_to_free_cpu()* routine through the use of the taskset command. According to [8], it may also be possible to set an environment variable, AFL_NO_AFFINITY = 1 to prevent a race condition from occurring, though we did not attempt this.



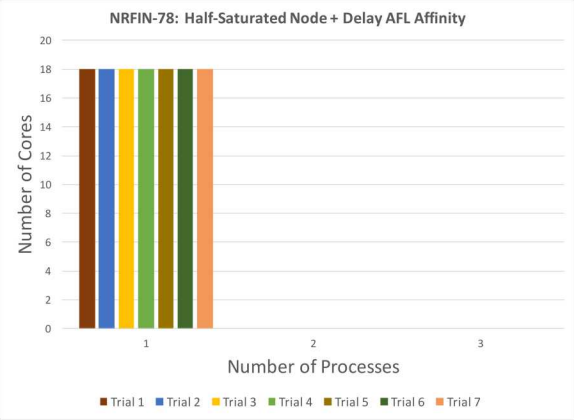
(a) Basic AFL Affinity



(b) Delay



(c) Half-Saturated Node



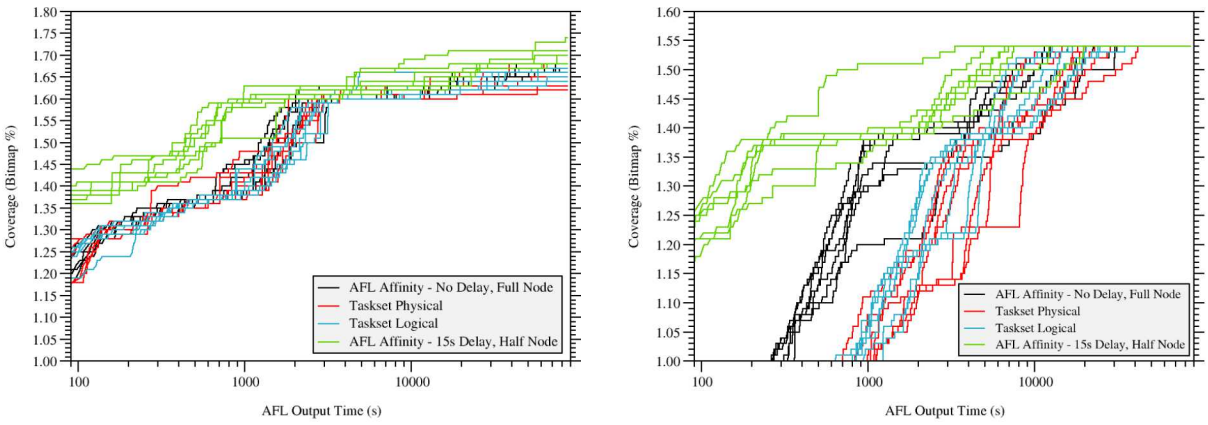
(d) Half-Saturated Node + Delay

Figure 4-2. Plots of the number of fuzzing instances (i.e. processes) bound to available cores for NRFIN-78 run with AFL Affinity. Using the default AFL settings (Figure 4-2(a)&(c)) leads to processor contention. The inclusion of a time delay between concurrent AFL execution calls fully resolves the issue, as can be seen in Figure 4-2(b)&(d).

4.2. CPU Binding Technique

Utilizing a full node with all processors fuzzing the target binary in a coordinated fashion, we fuzzed the CROMU-9 and NRFIN-78 binaries using all three defined types of running AFL. Figure 4-3 shows the composite coverage for all 21 experiments (three types * seven trials each) as a function of wall time for each binary. Here, the data plotted is an aggregate of all 36 fuzzers. When AFL is run in a coordinated fashion, there is one master and (in our case) 35 slave fuzzers. Each fuzzer keeps a local log of its coverage progress, and every so often the master fuzzer syncs its log with all slaves. As a result, if one simply ignores the intermediate progress of all slaves and only plots coverage as a function of execution time for the master fuzzer, the resulting data can be quite misleading. Instead, we combined the coverage logs for all fuzzers into a single file and, when duplicate time values were encountered, we kept only the largest coverage value.

Utilizing a half node (eighteen physical cores) with all processors fuzzing the target binary in a coordinated fashion, we ran AFL Affinity with a time delay of fifteen seconds between concurrent calls on both the CROMU-9 and NRFIN-78 binaries. The results are depicted as solid green lines in Figure 4-3. This combined choice of adding a time delay between concurrent AFL calls (to resolve the processor contention effect) along with only utilizing half of the available physical cores led to a notable boost in coverage exploration early in wall time. In the case of CROMU-9, the final coverage after 24 hours was also higher (median value).

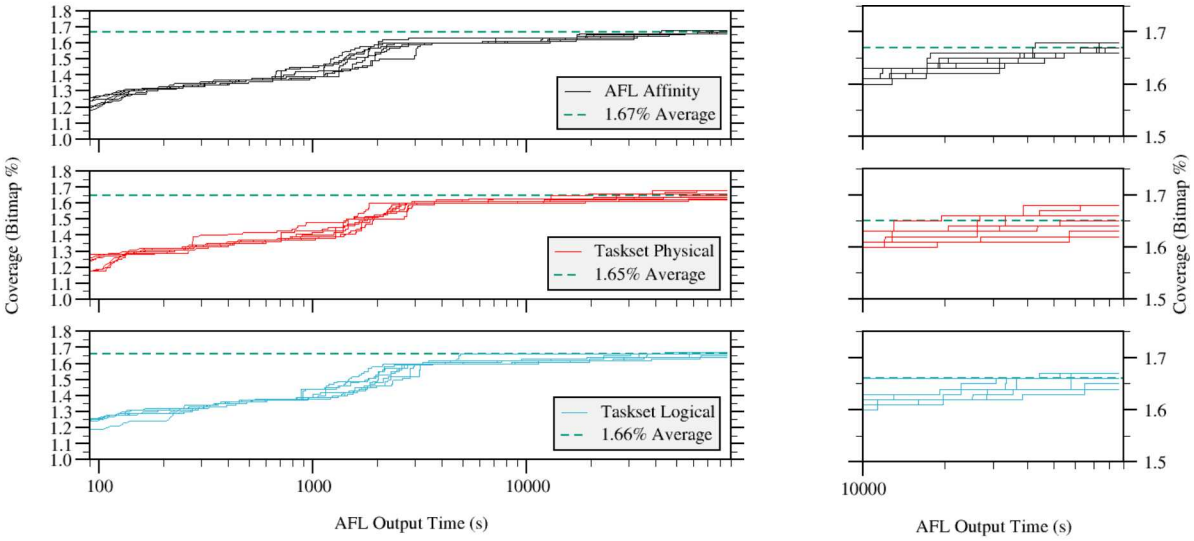


(a) CROMU-9 (b) NRFIN-78

Figure 4-3. 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.

Figure 4-4 presents a stacked version of the data in Figure 4-3(a) for CROMU-9, as much of the coverage overlaps (especially towards the end of the 24-hour period) and is thus difficult to decipher with precision. Close visual inspection indicates that whether only a single physical core (Taskset Physical) or a combination of a physical plus logical core pair (Taskset Logical) are used, the final coverage after 24 hours is nearly the same. This outcome is repeatable, as indicated by the tighter range of final code covered as compared with AFL Affinity. Indeed, the

mean final coverage (green horizontal lines in Figure 4-4) is identical between Taskset Physical and Taskset Logical for a set of seven trials. Interestingly, the mean coverage of the seven trials of AFL Affinity is quite like the two Taskset types, despite the erratic behavior of over-assigning processes to cores. Nonetheless, this outcome is not always guaranteed and may be dependent on the binary that is fuzzed, the length of time the fuzzer is allowed to run, and other factors. We therefore recommend running AFL Affinity with a time delay between concurrent calls to resolve the uneven workload distribution problem. Alternatively, running AFL with either Taskset Physical or Taskset Logical should equally resolve the processor contention issue albeit with a bit of extra work.



(a) Data for the full run, up to 86,400 seconds. (b) The final decade.

Figure 4-4. A stacked version of the data from Figure 4-3(a), for the CROMU-9 binary. For each type, the mean final coverage from the seven trials is presented as a dark green dashed horizontal line. Note that the final mean coverage is identical to the median value, rounded to two decimal places.

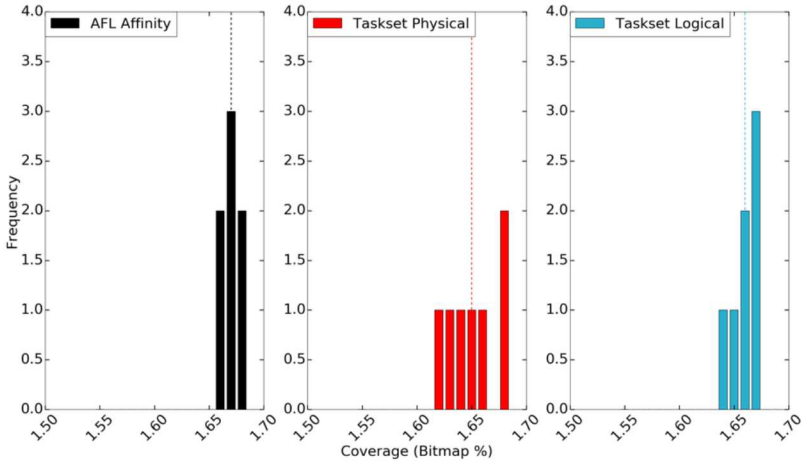


Figure 4-5. A histogram of the final code coverage frequency for the set of seven trials for each AFL type after fuzzing the CROMU-9 binary for 24 hours. The vertical dashed lines represent the median value of the dataset.

The distribution of final coverage achieved after 24 hours of fuzzing the CROMU-9 binary is presented as a histogram in Figure 4-5. The vertical dashed lines in the graphic represents the median value of each dataset.

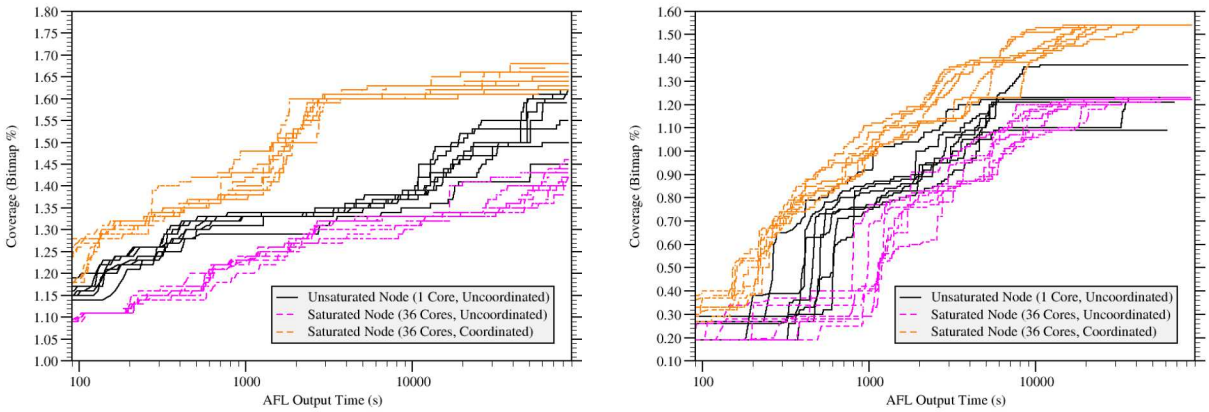
4.3. Effect of Node Saturation and Coordination

4.3.1. CROMU-9

We investigated the effect of saturating a single node with AFL tasks that were both coordinated and uncoordinated. Three experiments were performed: (i) a single AFL instance on one HPC core (thus uncoordinated and unsaturated), (ii) 36 unique AFL instances running on 36 cores (uncoordinated, saturated), and (iii) 36 combined AFL instances (one master/35 slaves) running on 36 cores (coordinated, saturated). AFL type Taskset Physical was exclusively implemented for this series. Seven trials for each experiment were performed. Figure 4-6(a) depicts the coverage as a function of wall time for this series.

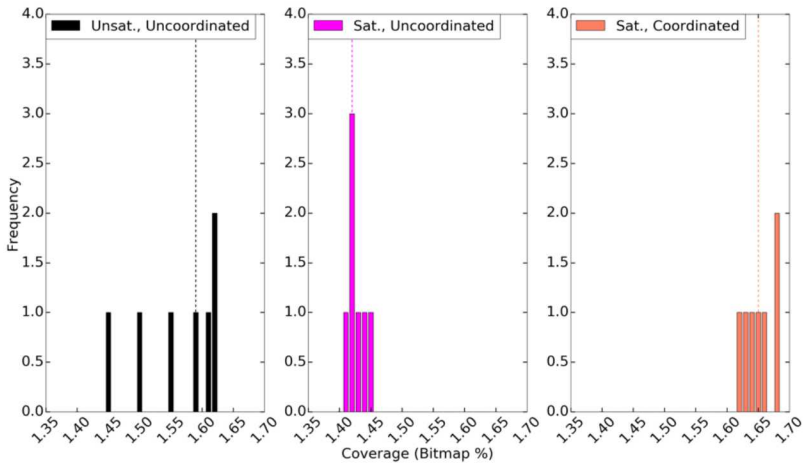
The outcome is largely as expected. The worst performance is observed when a node is fully loaded with independent AFL tasks whose efforts are uncoordinated (pink dashed lines). While this scenario is the ideal case from a job-packing efficiency perspective, it is unwise to load a node to capacity with independent tasks if coverage within a short time period is essential. The next best performer is the set of single core AFL tasks (black solid lines). Here we have one core on the node (specifically core #0) working on a task, while the remaining 35 cores sit idle for the duration of the experiment. Lastly, the best performance is observed when a fully packed node of coordinated AFL tasks are run (orange dashed lines). Even for the worst performer of this set, with a final coverage of 1.62%, 98.7% of the total coverage was achieved within the first 75 minutes of execution via this coordinated method.

Concerning the variability of the total coverage, similarly as above, we can depict the final coverage for the set of trials as a histogram to visually inspect the distribution. This is presented in Figure 4-7.

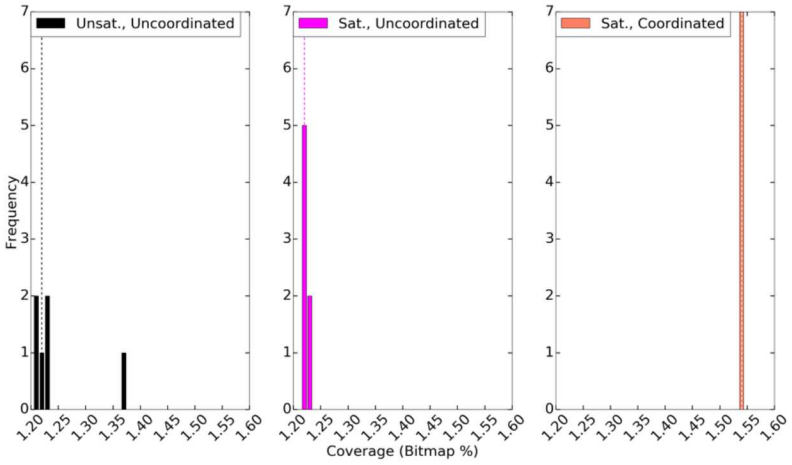


(a) CROMU-9 (b) NRFIN-78

Figure 4-6. Composite plot of the coverage as a function of AFL output time for Taskset Physical. For the "Saturated Node, Uncoordinated" dataset (magenta) the median performer for each trial, as measured by the final coverage from all 36 independent tasks, was selected for plotting.



(a) CROMU-9



(b) NRFIN-78

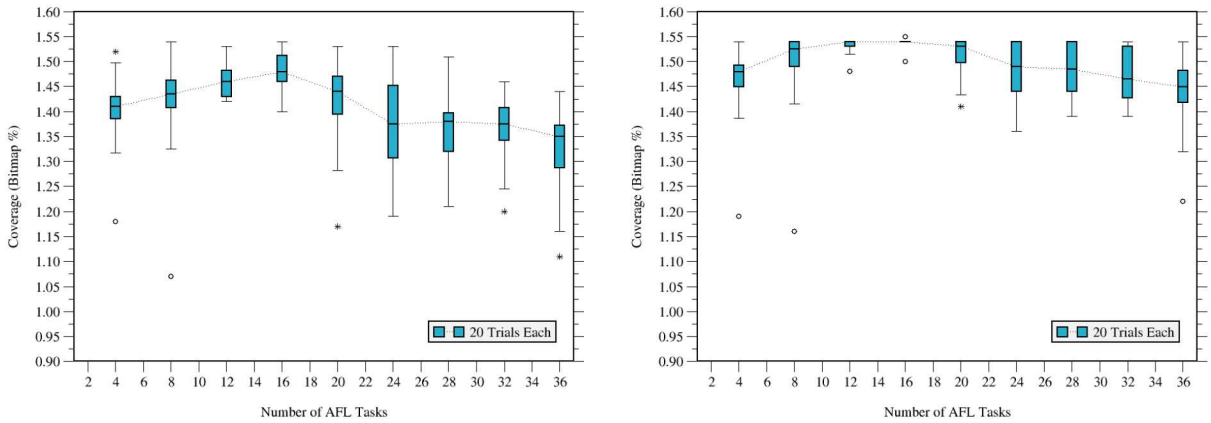
Figure 4-7. A histogram of the final coverage frequency for the trials of Taskset Physical experiments from Figure 4-6 for both the CROMU-9 and NRFIN-78 binaries. The vertical dashed line represents the median value of each dataset.

4.3.2. NRFIN-78

For the NRFIN-78 binary, we investigated the effect of node load to discover if there is an optimal number of cores to activate for fuzzing on a single node. Recall that our compute nodes have dual sockets with 18 cores each, for a total of 36 cores per node and 72 threads in full.

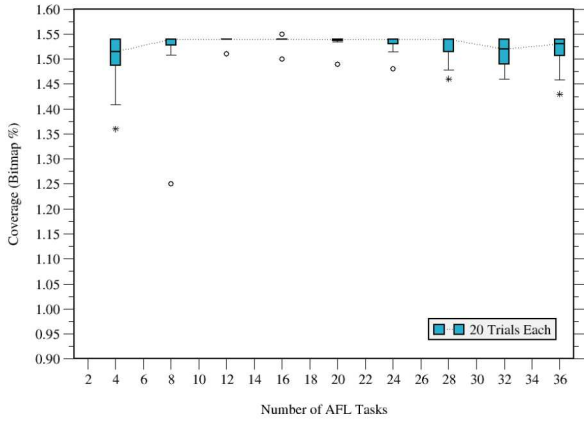
We ran AFL with default settings (AFL Affinity), and added a fifteen second pause between concurrent AFL calls to eliminate the effect of oversaturating cores with tasks. For this experiment, we increased the number of trials from seven to 20, and fuzzed the NRFIN-78 binary in all cases for 24 hours. A box and whisker plot of the coverage as a function of the number of AFL tasks (i.e., node load) was generated and is presented in Figure 4-8. Snapshots at hourly intervals were created, and we showcase the results after two, six, twelve, and 24 hours of coordinated fuzzing to illustrate the progression of coverage as a function of node load over time.

Notice that, early in time (after two hours (Figure 4-8(a)) and six hours (Figure 4-8(b))), the median of the boxes, as indicated by the dotted black trend line drawn through each box's median value, is greatest when sixteen tasks are used. For our hardware, recall that eighteen cores correspond to one socket, though, in this test, we chose to request multiples of four cores from four to 36. It is possible that, if the experiment were repeated with eighteen cores (tasks), that median value would in fact be the greatest median point.

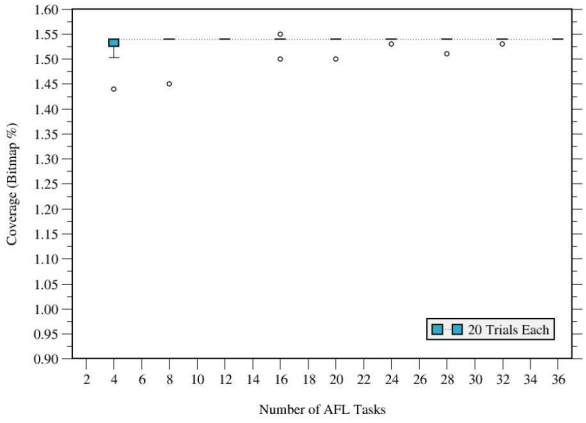


(a) After 2 hours.

(b) After 6 hours.



(c) After 12 hours.



(d) After 24 hours.

Figure 4-8. 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.

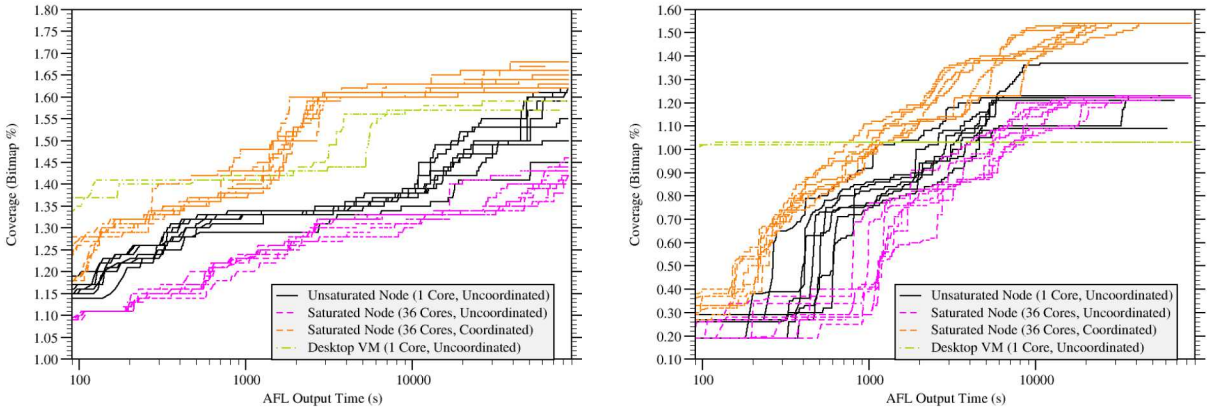
As the fuzzing progresses, the trend line of median values begins to flatten, and after 24 hours the boxes are all but collapsed for four AFL tasks (leftmost box in Figure 4-8(d)). Due to the high variability across fuzz runs, we have not determined with statistical significance the optimal number of cores to invoke. But we believe, absent any other information about the binary, $num\_cores/2$ should be chosen due to the peak we see in the median.

This observed effect may be due to the hardware configuration (dual sockets) and Intel's management of memory between them. The fact that performance levels off after $num\_cores/2$ causes us to hypothesize that this is a memory contention issue that we have seen before with these Xeon HPC machines on other tasks. If this is a problem, a potential solution would be to allocate AFL's memory to the same core that AFL is bound to. This requires further investigation.

4.4. Desktop Fuzzing

In addition to fuzzing on the HPC cluster, we briefly explored fuzzing both binaries with AFL on a desktop machine. A single-core virtual machine running Ubuntu Linux was used for this test. All else equal, we performed only two trials for this investigation. After 24 hours, the final coverage achieved for the CROMU-9 binary was 1.57% and 1.59%, and is shown in Figure 4-9(a). Compared to the black lines in Figure 4-6, these values fall at the high end of the HPC analogs at the end of 24 hours. This result suggests that running an isolated (single-core) instance of AFL is best suited for a dedicated standalone desktop machine rather than an HPC node (on average). The added benefit of such a setup would not waste additional cores that could potentially be utilized for other tasks.

The experiment was repeated using the NRFIN-78 binary on the same desktop machine, and the final coverage results for both trials was 1.03% as illustrated in Figure 4-9(b).



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

Figure 4-9. A repeat of the data in Figure 4-6, with the results of desktop fuzzing included as lime green dot-dash lines. Only two trials were used for the desktop fuzzing case.

5. AUTOMATING THE WORKFLOW WITH BLUECLAW

Manually generating the testbed composed of three different AFL execution types, in both serial and parallel, along with seven (or twenty!) different trials for each type, has proven to consist of a healthy amount of work. However, the task is amenable to scripting and thus we created a tool to accomplish this. 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 make it very flexible and robust, tailoring the functionality to a user's wants and requirements. The initial release is Version 1.0. Please contact Sandia National Laboratories for access.

5.1. Operation

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).

The following command will programmatically create an environment for AFL Affinity to fuzz a single binary for 24 hours in a coordinated fashion, assuming the AFL installation directory is in one's path:

```
blueclaw.py -t 24 -i 36 ./bin/CROMU_00009/ PS
```

In this case, we do not specifically direct BlueClaw to an AFL installation directory. It is assumed that AFL is installed and has been added to the user's path environment. The user directed BlueClaw to the directory ./bin/CROMU_00009/ as opposed to the actual binary (…/file.exe). This is for convenience, but the user should be cautious and certain that only one executable file resides in the provided directory.

The following command will create an environment like above, but for Taskset Physical:

```
blueclaw.py -t 24 -i 36 -a ../afl-2.52b-comment/afl-fuzz ./bin/CROMU_00009/ PS
```

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 to fuzz are located must be a relative path.

The following command will generate an environment for Taskset Logical, where a combination of physical and logical cores will be activated for parallel (coordinated) fuzzing:

```
blueclaw.py -l -t 24 -i 36 -a ../afl-2.52b-comment/afl-fuzz ./bin/CROMU_00009/ PS
```

Note the only difference between the above command and that which activates Taskset Physical is the presence of the -l option. As mentioned, activating logical cores is only valid for parallel fuzzing. There are checks within the script to ensure that (i) the hardware is capable of this request, and (ii) the user is indeed requesting a parallel job.

A helpful feature of BlueClaw is the ability to inherently scale across multiple nodes. On the SRN cluster Ghost, there are 36 cores per node, but if we request 37+ fuzzing instances in mode

PS (or alternatively wish to fuzz 37+ target binaries in mode SM), the resulting job submission script will request two nodes where the first node is fully saturated with work (36 cores) and the remaining tasks (here, one) are spilled over to the next node in the list of available reserves.

5.2. Implementation

BlueClaw seeks to be as generic and flexible to its environment as possible. It contains scanning routines to find all binaries for fuzzing in a target directory, checks for hyperthreading, checks for available number of cores and nodes, and will react accordingly to what it encounters. If fuzzing is to be performed in a coordinated fashion, the script will generate the appropriate syntax for master/slave mode. If fuzzing is to be performed in an uncoordinated fashion, a single node will be populated with AFL tasks before spilling over to the next available node (if more than one node was requested). Note that while efforts have been made to be generic and flexible, BlueClaw has only been utilized on the SRN HPC resources as there are some hardcoded environmental assumptions, such as the location of shared directories and installed packages.

The job scheduler in use on the SRN HPC resources is the Slurm Workload Manager. Users submit jobs to the queue via the "sbatch" command. This functionality, along with the creation of job submission scripts, is built into BlueClaw. Depending on the flags passed to BlueClaw, a unique submission script is generated and the logic needed to correctly communicate with the Slurm scheduler is laced into the resulting "submit.sh" file.

5.3. Future Features

We plan to add support for a statistics option that will run *N* number of trials for a given experimental configuration. Another planned feature is a "dependent jobs" option that will allow a user to submit a request for longer than the maximum wall time allowed by the HPC scheduler. In that case, one would specify the total wall time needed (for example, 720 hours) and the submission script would be broken up into *M* scripts, each requesting the maximum allowed number of hours until the total requested wall time is reached.

One additional planned feature is a fourth run mode: PM, or **P**arallel execution against **M**ultiple binaries.

6. CONCLUSION

We have defined and presented three different AFL execution types. AFL Affinity is standard AFL, directly out of the box. Taskset Physical and Taskset Logical are slightly modified versions of AFL which use the taskset Linux command line tool to manually bind AFL processes to cores, rather than utilizing AFL's internal binding routine. While Taskset Physical uses only physical cores, Taskset Logical extends this to include a pair of physical and logical cores (if hyperthreading is available). We fuzzed the Cyber Grand Challenge binaries CROMU-00009 and NRFIN-00078 for 24 hours on the Sandia Restricted Network HPC cluster Ghost using the three different types of execution.

We further investigated the effect of saturating a node (in our case, 36 physical cores per node) with AFL instances running in both a coordinated and uncoordinated fashion. Results indicate that the most productive (greatest coverage after 24 hours) method of running AFL is via AFL Affinity with a time delay between coincident AFL calls and utilizing ~50% of available physical cores to fuzz a target binary in a coordinated fashion (lime green lines in Figure 4-3). Not only did this method produce the most coverage in 24 hours, but much of the coverage occurred within the first 75 minutes via this method. Whether this is an anomaly of Intel's method of memory management between sockets is unknown. Further testing on differing architectures would have to be performed to make a final determination.

Lastly, a Python script named BlueClaw was created to automate the tedious task of establishing a testbed for experimental fuzzing on an HPC system. Through a small handful of flexible command line options, a user can very quickly establish a suite of fuzzing experiments against any number of binaries. What results from each execution is a job submission file that is ready to be submitted to the Slurm job scheduler. BlueClaw is designed to be user friendly and fluidly scales across any number of nodes.

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

[6] R. M. Love, "taskset(1) - Linux man page," die.net, [Online]. Available: https://linux.die.net/man/1/taskset. [Accessed 23 05 2018].

[7] Sandia National Laboratories, "Sandia National Labs High-Performance Computing," 2018. [Online]. Available: http://hpc.sandia.gov/platforms/index.html. [Accessed 30 04 2018].

[8] Gamozo Labs, "Scaling AFL to a 256 thread machine," 16 09 2018. [Online]. Available: https://gamozolabs.github.io/fuzzing/2018/09/16/scaling_afl.html. [Accessed 04 08 2018].

DISTRIBUTION

Email—Internal

Name	Org.	Sandia Email Address
Nasser Salim	5831	njsalim@sandia.gov
Christian R. Cioce	5836	crcioce@sandia.gov
Danny Loffredo	5838	dloffre@sandia.gov
Technical Library	9536	libref@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.