An Investigation on Improving Distributed Fuzzing

Submitted in partial fulfillment of the requirements for

the degree of

Master of Science

in

Information Security

Sears K. Schulz

B.S., Cyber Science, United States Air Force Academy

Carnegie Mellon University
Pittsburgh, PA

April, 2022

Acknowledgements

First and foremost, I would like to thank my thesis advisor, Dr. Maverick Woo, for his invaluable dedication to my education over the course of the past year. He has always had my best interest in mind and inspired me to go above and beyond in and outside the classroom.

I would like to thank the readers, Mihai Dutescu and Dr. Rohan Padhye, for their time and for providing crucial feedback to support my development.

I would like to thank the Siemens FLUFFI team of Roman Bendt, Abian Blome, Mihai Dutescu, and Thomas Riedmaier for providing the essential knowledge and resources needed to complete my thesis.

I would like the thank the U.S. Air Force and the Air Force Institute of Technology for supporting my extended time at Carnegie Mellon University so that I could complete this thesis.

Finally, I would like to thank the Information Networking Institute for funding my education while serving in the U.S. Air Force, allowing me to continue pursuing my passion for the past two years.

This work was supported by the Information Networking Institute at Carnegie Mellon University, the Siemens Corporation, and the U.S. Air Force. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the sponsors.

Abstract

As software becomes more extensive and complex, identifying and remitting potential vulnerabilities is increasingly challenging. *Fuzzing* is an automated technique to discover bugs by repeatedly supplying the *program-under-test* (PUT) with generated inputs intended to trigger unknown bugs in the PUT. In 2016, Böhme et al. introduced the concept of *power schedules* and an improved *search strategy* to the then state-of-the-art fuzzer AFL. Using their implementation, which they dubbed AFLFast, they found that these changes resulted in significantly faster discovery of more crashes than AFL. In independent work, researchers at Siemens have been investigating how to take advantage of data center scale infrastructure best when fuzzing. To encourage adoption and facilitate academic research, they have open-sourced their own *distributed fuzzing* system, FLUFFI, in September 2019.

This thesis investigates the application of the power schedule and search strategy in AFLFast to FLUFFI. Specifically, we have implemented AFLFast's power schedule and search strategy as well as a round-robin search strategy on top of the upstream version of FLUFFI. To evaluate the effectiveness of these changes, we have chosen 10 binaries with known bugs from Google's FuzzBench and measured the differences in code and bug coverage between different combinations of power schedules and search strategies. Our findings include: (i) the ideas of Böhme et al. can be applied to FLUFFI to improve the fuzzing outcomes in a manner similar to how AFLFast improved upon AFL; and (ii) despite its simplicity, round-robin can be a desirable search strategy earlier in a fuzzing campaign.

Table of Contents

List of Tables

List of Figures

Symbols

$\alpha(i)$	The number of energy the original AFL implementation assigns for seed i.
β	A constant used in a power schedule.
$f(i)$	The number of seeds that exercise the same path as seed i.
$p(i)$	The number of energy, or new seeds to be generated, for seed i.
$s(i)$	The number of times the seed i has been chosen from the queue.
M	A constant that defines the maximum number of energy a power schedule can assign.

Abbreviations

AFL	American Fuzzy Lop; an open-source evolutionary fuzzer.
ASLR	address space layout randomization; a technique to randomize addresses at runtime to prevent memory corruption.
DBI	dynamic binary instrumentation; a technique that involves modifying instructions of a binary during execution.
FAST	the exponential power schedule used in AFLFast.
FLUFFI	Fully Localized Utility For Fuzzing Instantaneously; a distributed fuzzer developed by Siemens.
PUT	program-under-test; the software being tested by a fuzzer.
TLS	thread-local storage; global memory local to a thread.

1

Introduction

As the size of an organization's software grows, it can become increasingly challenging to ensure it has minimal bugs or vulnerabilities before deployment. *Fuzzing* is a well-known technique that is the real world to discover software vulnerabilities. It involves repeatedly running the *program-under-test* (PUT) with concrete inputs generated algorithmically, such as based on prior outputs or code coverage. Fuzzing has become popular due to its simplicity - it requires minimal configuration and is fully automated. Implementations such as the established American Fuzzy Lop (AFL) have successfully discovered numerous software vulnerabilities.

In general, the more inputs tested against the PUT, the more likely vulnerabilities will be discovered. Naturally, testing numerous inputs requires time and processing power. One method to increase processing power is *distributed fuzzing*, which involves a fleet of multiple machines running in parallel to fuzz the PUT. The challenges of distributed fuzzing include determining which inputs each machine will test and how they will communicate their results within the server fleet.

A successful distributed fuzzer should be able to spread a fuzzing workload across numerous machines, such as a large data center or cloud resources, with minimal configuration. To ensure work is not duplicated, each node should run different inputs against the PUT. Since fuzzing is a relatively new field, distributed fuzzers should ideally implement the most recent advancements in fuzzing.

AFL and its more advanced fork, AFL++, feature main and secondary modes to support distributed fuzzing. A directory is established to sync the fuzzer state. Main nodes sync the input queue with all nodes, while the secondary nodes only sync with the main node. To distribute fuzzing on more than one machine, the developers recommend periodically transferring the sync directories via SCP [6].

While AFL++ does incorporate recent fuzzing research, its distributed functionality is lacking. Users need to write their own scripts to compress and transfer the sync directories between machines. Furthermore, since the nodes sync periodically, work between nodes may be duplicated due to delays in receiving new information. Nodes on the same machine are synced every 30 minutes, and the developers recommend syncing between remote machines every 60 minutes.

Fully Localized Utility For Fuzzing Instantaneously (FLUFFI) is a distributed fuzzer built by Siemens to take advantage of its large data centers. From the beginning of its development, the fuzzer was designed to be a distributed system. Nodes in the system receive work by querying a central database such that no work is replicated. FLUFFI was open-sourced in September 2019 to encourage contributions from the academic community.

While FLUFFI's method for fuzzing is similar to that of AFL, it does not embrace more recent fuzzing advancements as AFL++ does. One example is AFLFast, a fork of AFL that implements a power schedule and new search strategy. The authors of AFLFast find that the fork outperforms AFL in code and bug coverage [4]. This thesis investigates the application of AFLFast's power schedule and search strategy to FLUFFI. We have evaluated various combinations of power schedules and search strategies on 10 PUTs with known bugs from Google's FuzzBench.

We find that: (i) the ideas of Böhme et al. can be applied to FLUFFI to improve the fuzzing outcomes in a manner similar to how AFLFast improved upon AFL; and (ii) despite its simplicity, round-robin can be a desirable search strategy earlier in a fuzzing campaign. This thesis makes the following contributions:

- We introduce and evaluate a power schedule in a distributed fuzzer to control the number of inputs generated from a seed.
- We introduce and evaluate two new search strategies in a distributed fuzzer to influence which seeds are prioritized for fuzzing.
- We show that the ideas of Böhme et al. 2016 [4] can be transferred to a distributed fuzzer.
- We show that a fuzzer's optimal strategy depends on the relative time position in the campaign.

2

Background

2.1 Fuzzing

Fuzzers can generally fall into one of three categories: *blackbox*, *whitebox*, and *grey-box* fuzzers. Blackbox fuzzers cannot observe the internals of the PUT, so they must rely on only the input and output of running the binary. On the contrary, whitebox fuzzers analyze the internals of the PUT. This extra information allows for dynamic symbolic execution, or concolic execution, which means that both concrete and symbolic inputs are used during execution. Due to increased information to analyze, whitebox fuzzers typically introduce more overhead than blackbox fuzzers. A compromise between these two options is a greybox fuzzer, which uses a smaller amount of information from the PUT internals. This analysis tends to be an approximation as it favors throughput. Greybox fuzzers often instrument the PUT at compile-time or use *dynamic binary instrumentation* (DBI) to modify instructions during execution. This instrumentation of the PUT provides more output metrics, such as code coverage [12].

Figure 2.1 depicts a simplified process of how a greybox fuzzer operates. First,

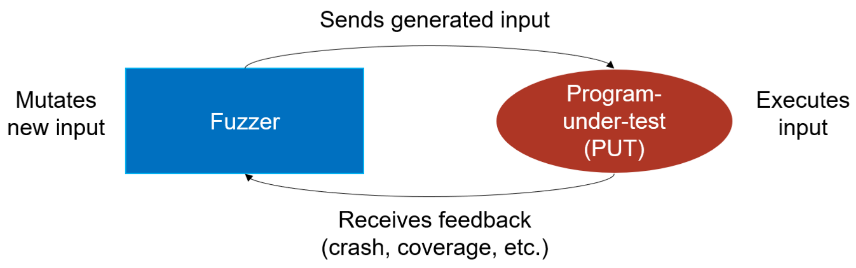

Figure 2.1: General strategy for greybox fuzzing.

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

Figure 2.2: AFL instrumentation [4].

the fuzzer sends an input, or a *seed*, to the PUT. After the PUT executes the seed, the fuzzer receives feedback about its execution, such as whether it crashes and the code coverage. Based on this information, the fuzzer generates, or *mutates*, more seeds from an existing seed. This process continues in a loop [12].

2.2 AFL

AFL is one of the most well-known fuzzers and is the basis for many of its successors. AFL is a greybox fuzzer. PUTs can be instrumented using the included compiler, `afl-cc`. The compiler instruments the binary by injecting additional instructions at the end of each basic block. The pseudocode is shown in Figure 2.2.

`cur_location` is the address of the current basic block (usually an offset from the start of the current module). `shared_mem[]` is a 64 kB shared memory region. This memory region stores a counter of each transition from one basic block to another. Since `prev_location` is not simply a copy of `cur_location` because of the added shift operation, the transition from A to B will be counted differently as the transition from B to A. The shared memory region is hashed after the PUT finishes execution. The hash can imperfectly represent a path in the PUT's execution [4]. Hash collisions could occur, or different basic block transitions may share the same index in the shared memory region.

Algorithm 1 AFL strategy [4].

```
Input: Seed Inputs $S$
 1: $T_X = \varnothing$
 2: $T = S$
 3: if $T = \varnothing$ then
 4:     add empty file to $T$
 5: end if
 6: repeat
 7:     $t$ = chooseNext($T$)
 8:     $p$ = assignEnergy($t$)
 9:     for $i$ from 1 to $p$ do
10:         $t'$ = mutateInput($t$)
11:         if $t'$ crashes then
12:             add $t'$ to $T_X$
13:         else if isInteresting($t'$) then
14:             add $t'$ to $T$
15:         end if
16:     end for
17: until timeout reached or abort-signal
Output: Crashing Inputs $T_X$
```

The general decision flow for AFL is roughly portrayed in Algorithm 1. A set of initial seeds are provided, such as a known valid input. If the user provides no seeds, an empty file is used as the initial seed. These seeds S are added to the queue T. New seeds t to test are chosen from the queue that are considered *favorites*; those with the fastest and smallest input for the path in the PUT it exercises. assignEnergy determines the number of new inputs generated from the chosen seed. AFL considers execution time, path coverage, and creation time of the seed t. To generate new seeds, mutateInputs will perform operations on the original seed, including bit flips, arithmetic operations, boundary values, and block deletion/inserts. The isInteresting function will determine if a new seed t' should be added to the queue T after executing it. isInteresting will return true if t' exercises new basic block transitions. A "bucketing" technique prevents path explosion due to loops. Seeds that crash the PUT will always be considered interesting and are added to T_X [4].

Functions of Interest. In Algorithm 1, the two functions of interest for this investigation are chooseNext and assignEnergy. chooseNext determines the *search strategy*; which seeds are prioritized when choosing a new one from the queue. assignEnergy determines the *power schedule*; how many new seeds should be generated from the seed chosen from the queue.

2.3 AFLFast

AFLFast is a fork of AFL that primarily alters the search strategy and power schedule of AFL. The authors, Böhme et al., found that AFLFast performed significantly better than AFL in code coverage and finding crashes. For example, AFLFast was compared to AFL using the 150 binaries from the Cyber Grand Challenge. Not only did AFLFast find crashes 19 times faster than AFL, but it also found seven crashes that AFL never discovered [4].

2.3.1 Markov Chain Model

The authors attribute AFLFast's success to modeling greybox fuzzing as a Markov chain. A Markov chain is a set of states with transitions between them. Each transition has a probability of it occurring. Böhme et al. model a potential path in a PUT as a state. Transitions occur when a seed exercises a different path than the seed it was generated from. Figure 2.3 shows an example of a Markov chain for a simple

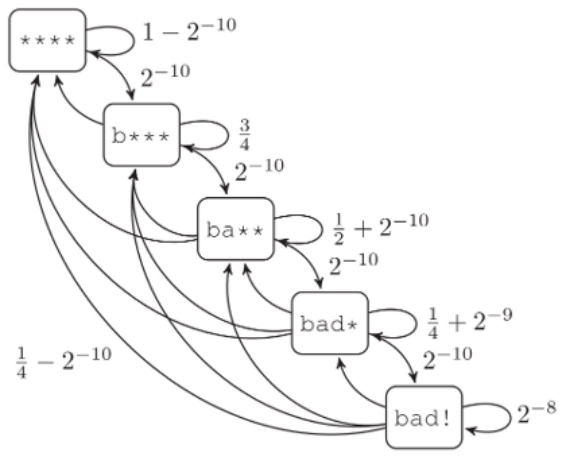

Figure 2.3: Markov chain for example program [4].

program. The program checks if the input is equal to `bad!` character by character. It is assumed that the inputs are always four ASCII characters. The Markov chain shows the probability of transitioning from one program path to another. For example, the current input may be `bill`, so the current state is `b***`. To transition to the `ba**` state, the correct character position to mutate must be chosen. Choosing the second position is a $\frac{1}{4}$ probability. Next, the correct character must be generated. Assuming 2^8 ASCII characters, the probability of choosing an `a` is 2^{-8}. Multiplying these two probabilities results in 2^{-10}, which is the probability of discovering the `ba**` path from the `b***` path.

p_{ij} is defined as the probability that mutating a seed that exercises path i will result in a new seed that exercises path j. *Energy* is the number of seeds that are generated by mutating another. By the linearity of expectation, to discover path j, the *minimum energy* is p_{ij}^{-1}. In this example, the minimum energy to discover the `ba**` path from the `b***` path would be 2^{10}. While p_{ij} is unknown, it can be estimated using $f(i)$, the number of seeds that exercise path i [4].

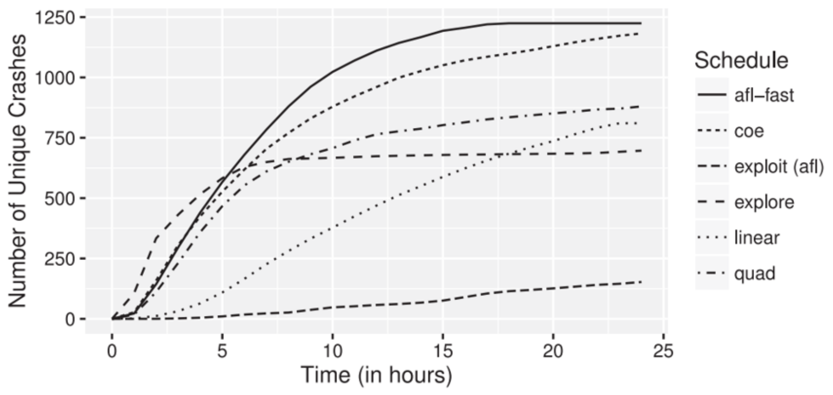

Figure 2.4: Comparing AFLFast power schedules by crash coverage [4].

2.3.2 Power Schedule

AFL's power schedule can be defined as:

$$p(i) = \alpha(i) \tag{2.1}$$

where $\alpha(i)$ is AFL's implementation of assignEnergy. AFL's power schedule outputs an energy that does not change as the number of times a seed is chosen from the queue, or $s(i)$. Böhme et al. propose five power schedules:

- EXPLORE: the AFL power schedule divided by a constant
- LINEAR: $p(i)$ is directly proportional to $s(i)$
- QUAD: $p(i)$ increases in a quadratic manner as $s(i)$ increases
- COE: $p(i)$ increases exponentially as $s(i)$ increases and goes to zero when $f(i)$ is greater than the mean $f(i)$
- FAST: $p(i)$ increases exponentially as $s(i)$ increases

Figure 2.4 compares the five power schedules and AFL in the number of unique crashes found. Böhme et al. find that the FAST power schedule is the most performant in this metric and other experiments. The reasoning for an exponential power schedule is that the minimum energy required to find a new path will be quickly reached while not generating too many seeds initially. The FAST power schedule is defined as:

$$p(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{2^{s(i)}}{f(i)}, M\right) \tag{2.2}$$

where β and M are constants [4].

2.3.3 Search Strategy

AFL selects seeds from the queue to mutate in a round-robin fashion. Seeds are chosen in the order they are added to the queue. Once all seeds have been fuzzed, AFL resumes with the first seed. New seeds are added to the back of the queue. AFLFast implements a greedy search strategy to fuzz new seeds and discover new paths. It always chooses the seed with the lowest $s(i)$. If there is a tie, it prioritizes the seed with the lowest $f(i)$. If there are still multiple contenders, the seed with the smallest and fastest input is selected [4].

2.4 FLUFFI

Fully Localized Utility For Fuzzing Instantaneously (FLUFFI) is a distributed fuzzer developed by Siemens. A version of the fuzzer is open-source and available at: `https://github.com/siemens/fluffi`. It was developed with several design choices in mind, such as:

- Only a binary is provided; no source code.

- A diverse set of targets, including architecture (x86, ARM), operating system (Windows, Linux), and interface (file, network, API).
- Limited time for developing a fuzzing harness.
- Throughput (executions per second) is not a priority [14].

2.4.1 Design

The overall architecture for FLUFFI is portrayed in Figure 2.5. The central server for the distributed system is called the *Global Manager*. It hosts a MariaDB database server, FTP server, DNS server, and a web interface. Using the web interface, users can create a *fuzzjob* representing a PUT's fuzzing campaign. Each worker server can host any number of *Local Managers*. Every fuzzjob is assigned one Local Manager. The worker servers can also host an arbitrary number of *agents*. Agents fall into three types: *Generators*, *Runners*, and *Evaluators*. The agents talk to their assigned fuzzjob's Local Manager to receive tasks. Since the agents are stateless, only the Local Manager queries the Global Manager's database server.

The general flow for a fuzzjob is shown in Figure 2.6. When more seeds are needed, the Generator will request a seed from the Local Manager. The Local Manager queries the Global Manager for a seed to be selected from the queue and sends it back to the Generator. The Generator mutates the seed to generate several new seeds. When Runners are ready, they will ask the Generator for a new seed to execute. Each Runner executes the seed on the PUT. The covered basic blocks list and the execution result (whether there was a crash, hang, etc.) are sent to an available Evaluator. Using a cache, the Evaluator checks whether new basic blocks were covered. The Evaluator will only tell the Local Manager to keep seeds that do not have a clean exit or find new basic blocks. If a seed is to be kept, the Evaluator will update the *rating* for the seed. The rating increases when the seed finds more

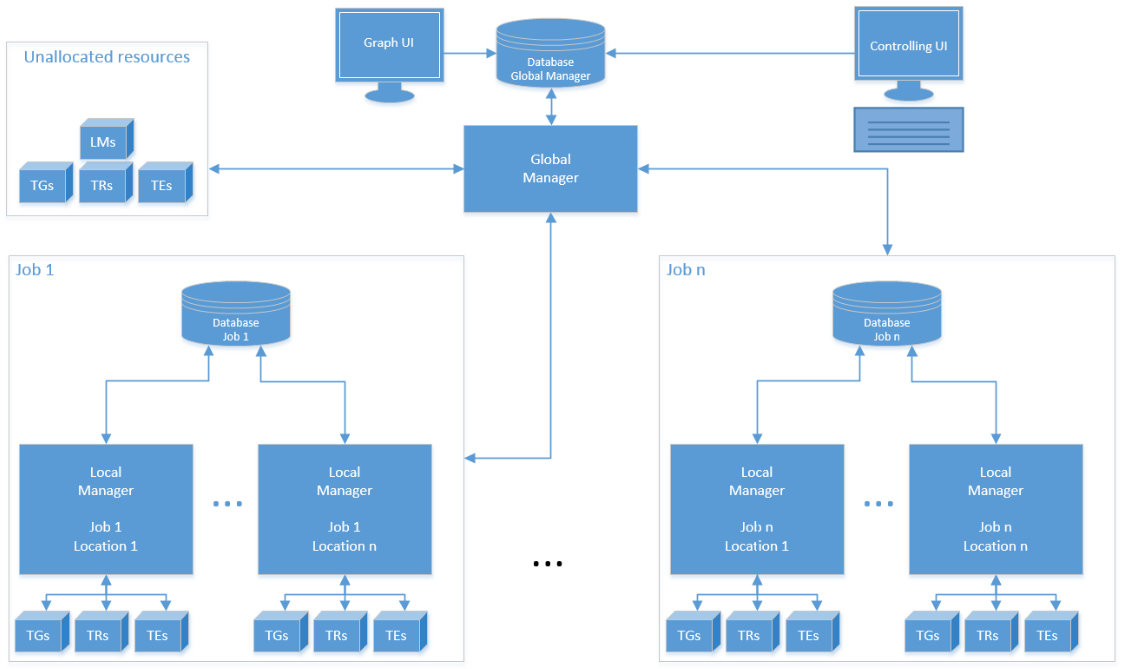

Figure 2.5: FLUFFI global architecture [14].

basic blocks and decreases when the process crashes, hangs, etc.

2.4.2 Mutators

One unique feature of FLUFFI is that multiple mutators can be enabled in the same fuzzing campaign. The five mutator options are AFL, Radamsa, Honggfuzz, and two internally developed ones. Unlike other fuzzers, one mutator can generate a seed using a seed generated from a different mutator. The parent of each seed is also tracked. Thus, the seeds' evolutionary tree can be viewed in the web interface. See Figure 2.7.

2.4.3 Dynamic Binary Instrumentation

Since FLUFFI does not instrument the PUT at compile-time, DBI is used to track execution coverage. Although this introduces a higher execution overhead,

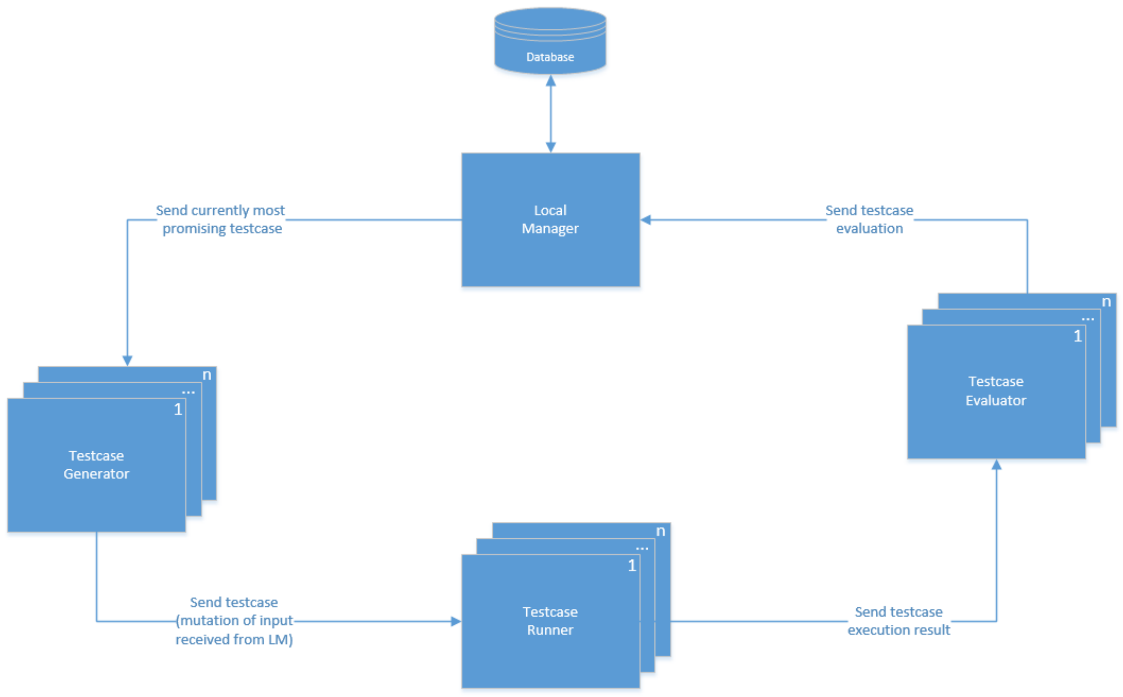
Database
Local Manager
Send currently most promising testcase
Send testcase evaluation
Testcase Generator
Testcase Evaluator
Testcase Runner
Send testcase (mutation of input received from LM)
Send testcase execution result
n
...
1

Figure 2.6: FLUFFI local architecture (note: the terms seed and testcase are used interchangeably) [14].

the PUT source code is not required, and fuzzing campaigns can be started effortlessly. FLUFFI incorporates DynamoRIO, an open-source DBI library (`https://github.com/DynamoRIO/dynamorio`). DynamoRIO copies the basic block into a cache when a basic block is first executed. See Figure 2.8. Only when DynamoRIO has added a basic black to a cache can it analyze the basic block or instrument it. To determine coverage, FLUFFI uses one of DynamoRIO's included clients, *drcov*. The client does not directly instrument the binary. It simply tracks when basic blocks are moved into the cache. After execution, it reports the addresses of the executed basic blocks. It does not track whether a basic block was executed more than once.

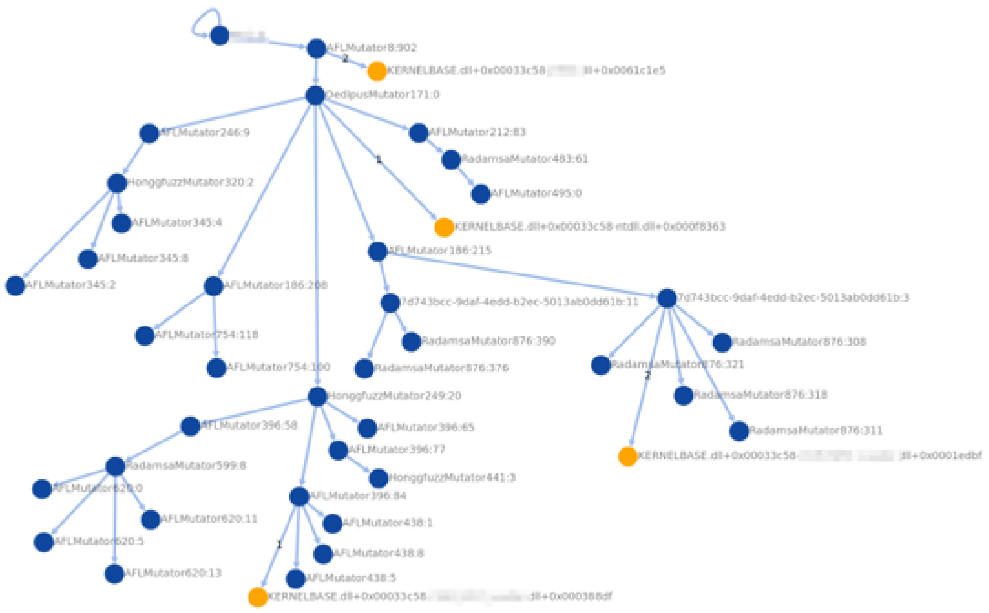

Figure 2.7: FLUFFI mutation tree example [3].

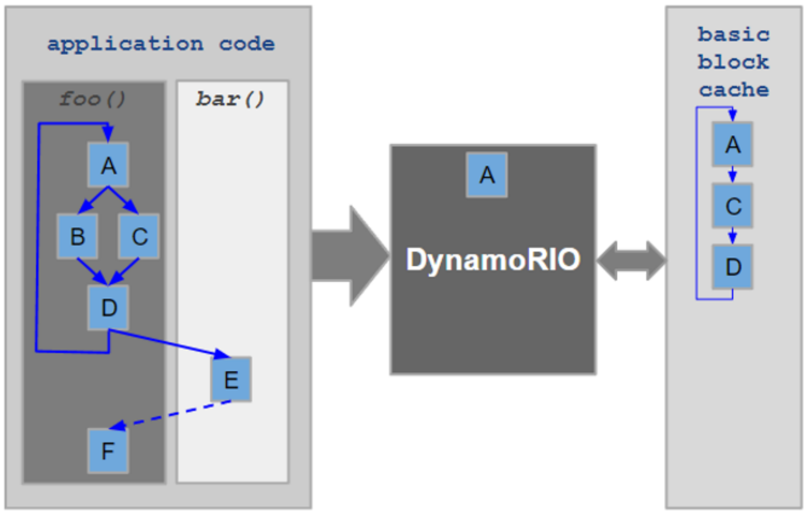

Figure 2.8: DynamoRIO dynamic binary instrumentation.

2.4.4 Characteristics

Rating. Each seed is assigned a rating. The rating represents how many more times the seed should be fuzzed. The rating increases when a child of the seed covers new basic blocks. The rating decreases when a child of the seed crashes or hangs in execution.

Power Schedule. FLUFFI implements a constant power schedule. When an Evaluator receives a seed to mutate, it will always generate 50 new seeds.

Search Strategy. FLUFFI implements a greedy search strategy. Interesting seeds are stored in the Global Manager's database and their assigned rating. When the Evaluator asks the Local Manager for a seed, the seed with the highest rating is always selected.

2.5 Fuzzing Experimentation

Evaluating and comparing fuzzers is a challenging task. The mutator introduces randomness, so one trial of a fuzzer could be significantly different from another trial. Distributed fuzzing further exacerbates this issue because network latency can introduce more randomness with numerous agents. Furthermore, the duration of a fuzzing campaign and the PUT can have a notable impact on its performance.

2.5.1 Klees et al. 2018

In *Evaluating Fuzz Testing*, Klees et al. recommend how fuzzers can adequately be evaluated. First, the authors claim that multiple trials should be run to sample the fuzzer's performance accurately. Additionally, a statistical test can then determine whether one fuzzer outperforms another. The authors recommend the Mann-Whitney U test, which compares two samples using ranked sums [10].

Second, Klees et al. argue that each trial should have a sufficiently long duration. While one fuzzer may outperform another at the beginning of a fuzzing campaign, the results could change over time. The authors recommend a duration of at least 24 hours because performance for shorter runs can be extracted from longer runs [10].

Third, the authors advocate for using a diverse set of PUTs. Considerations include varying sizes of the binary, input formats (document, video, text, etc.), and the type of initial seed(s). The PUTs should contain known bugs, so both bug and code coverage can measure that performance. Klees et al. recognize that there is currently no testing suite that meets their criteria [10].

2.5.2 FuzzBench

One attempt to develop a fuzz testing suite is Google's FuzzBench. The original paper uses 22 real-world programs with various input formats, size, and the number of initial seeds. None of the PUTs have known bugs, so performance is only measured by code coverage achieved by the fuzzer. FuzzBench compiles the PUTs and fuzzers in Docker containers. The experiment is also run in a Docker container. By default, FuzzBench runs 20 trials for each PUT with a duration of 23 hours. At the end of the experiment, a report is generated containing various statistics and graphs to compare the performance of the fuzzers [13].

One issue with the original FuzzBench suite is that none of the programs contained bugs, and performance is only measured by code coverage. FuzzBench has evolved since its original publication in an open-source repository (`https://github.com/google/fuzzbench`). Notably, new PUTs have been added that contain known bugs so that bug coverage can also be measured.

3

Design

3.1 FLUFFI

One conclusion from the AFLFast paper is that the addition of a power schedule and greedy search strategy to a fuzzer results in better fuzzing outcomes [4]. This experiment aims to determine if the same finding holds for FLUFFI in addition to AFL. To achieve the *transfer principle*, as many variables as possible must stay constant between AFLFast's experiment and this experiment.

Multiple combinations of power schedules and search strategies are deployed for a more comprehensive experiment, allowing for conclusions beyond simply replicating AFLFast. The following power schedules are implemented:

- Constant: 50 seeds are always generated (original FLUFFI implementation)
- FAST: the exponential power schedule used in AFLFast (see equation 2.2)

The following search strategies are implemented:

- FLUFFI: the seed with the highest rating is selected (original FLUFFI implementation)

ID	Power Schedule	Search Strategy
1	Constant	FLUFFI
2	FAST	FLUFFI
3	Constant	Round-Robin
4	FAST	Round-Robin
5	Constant	AFLFast
6	FAST	AFLFast

Table 3.1: FLUFFI implementations for experiment.

- Round-Robin: the seed that has not been chosen for the longest time is selected, and new seeds are added to the back of the queue
- AFLFast: the seed with the lowest $s(i)$, then lowest $f(i)$, then the highest rating is selected

The original search strategy for AFLFast used the fastest and smallest seed as the third parameter. This version of the AFLFast search strategy uses FLUFFI's rating as an alternative for the third parameter since FLUFFI does not track seed execution time.

Table 3.1 shows the implementations of FLUFFI needed for the experiment. Note that the ID of 1 is the original implementation of FLUFFI. The ID of 6 mimics AFLFast.

3.1.1 Code Coverage and Instrumentation

To implement AFLFast's power schedule and search strategy, $f(i)$, the number of seeds that exercise the same path, needs to be tracked. Therefore, there must be a method for FLUFFI to differentiate and count unique paths. FLUFFI currently tracks coverage using DynamoRIO's `drcov` client. Since basic blocks are only intercepted the first time they are moved into the cache, `drcov` returns a deduplicated list of covered basic blocks. This list is an option to represent paths. However, there

```
void bar(int num) {
  ...
  foo(num * 10);
  ...
}

bar(input * 100);

for (int i = 0; i < input; i++) {
  foo(i);
}
```

Figure 3.1: Example C program to demonstrate different paths with same basic block coverage.

are cases where a list of basic blocks would not be granular enough to distinguish different paths.

Figure 3.1 shows an example of how FLUFFI would be unable to differentiate between different paths in a C program. First, the input to the program affects the number of iterations the loop will run. A deduplicated basic list does not distinguish between various loop iterations. Second, `foo()` is called from `bar()` and within the loop. To `drcov`, there is no difference between the calls to `foo()`. These are just two examples portraying that the current implementation of FLUFFI's coverage collection is not as granular as desired.

AFL's instrumentation, called *edge coverage*, builds an array where each index represents a transition between two basic blocks, and the value counts how many times that transition has occurred. See Figure 2.2. This coverage method can distinguish between different loop iterations and track where a function was called from. Implementing AFL's edge coverage in FLUFFI would allow for more granular code coverage to keep track of unique paths.

To maintain the original functionality of FLUFFI, `drcov` should still be run on

```
MOV reg3, TLS;
MOV reg1, [reg3];
MOV reg2, shared_mem;
XOR reg1, offset;
INC [reg2 + reg1];
MOV [reg3], offset >> 1;
```

Figure 3.2: `drcov` instrumentation.

the PUT to receive basic block coverage. Additionally, basic block and edge coverage should be collected in a single PUT execution to reduce the overhead of the changes. To accomplish this, `drcov` is modified to report edge coverage and the original basic block coverage. WinAFL, a Windows fuzzer that uses DynamoRIO to collect edge coverage, was used as inspiration for developing the `drcov` patch [8].

Except for saving and restoring registers, Figure 3.2 shows the six instructions that DynamoRIO prepends to each basic block. The instrumentation only occurs on the PUT and not on any shared libraries. The previous and current locations are stored as offsets from the start of the PUT's memory space. This modification prevents address space layout randomization (ASLR) from affecting the edge coverage result. The previous offset is stored in thread-local storage (TLS), a global memory local to a thread. Storing the previous location in TLS allows for multithreaded PUTs to be supported. The previous offset is initialized to 0. The pointer to the previous offset is first moved into `reg3` and then dereferenced into `reg1`. The pointer to the shared memory array is loaded into `reg2`. The previous offset (`reg1`) and the current offset are XOR'd. The result of this operation is the index in the shared memory array that is incremented. Finally, the current offset bit shifted to the right is stored in TLS. The shared memory array is hashed and returned in `drcov`'s output.

3.1.2 Tracking Additional Data

FLUFFI needs to track various new data to support the new power schedule and two new search strategies. The Global Manager's SQL database acts as a central data repository for each fuzzing campaign. The database schema is modified to store the new data. A new edge coverage table with two columns is added. The primary key stores a hash of the edge coverage's shared memory array to represent a unique path. Each row also contains a counter, which is incremented each time a PUT execution exercises the same path.

The SQL database already contains a table for the seed queue, where each row is a seed. A column is added with a timestamp of when the seed was last chosen from the queue. This data is necessary for the round-robin search strategy. The timestamp is set to the current time when a new seed is added or chosen to be mutated. Another column is added that contains the number of times the seed has been selected from the queue, or $s(i)$. This counter initializes at 0 and increments when the seed is chosen to be mutated. Finally, a column is added with the PUT's edge coverage hash. Thus, the edge coverage table can be joined to obtain $f(i)$.

3.1.3 Message Changes

In FLUFFI, the data sent between the Local Manager, Generators, Runners, and Evaluators are called messages. Additional information is included in these messages to support tracking edge coverage and the FAST power schedule because only the Local Manager has access to the data in the SQL database. Figure 3.3 shows the fields that must be added to the messages.

The Runner executes the PUT with the modified `drcov` instrumentation that outputs the new edge coverage hash. The original implementation of FLUFFI sends the execution result to an available Evaluator, including whether the PUT crashed and the basic block coverage. The edge coverage hash is added to the content of this

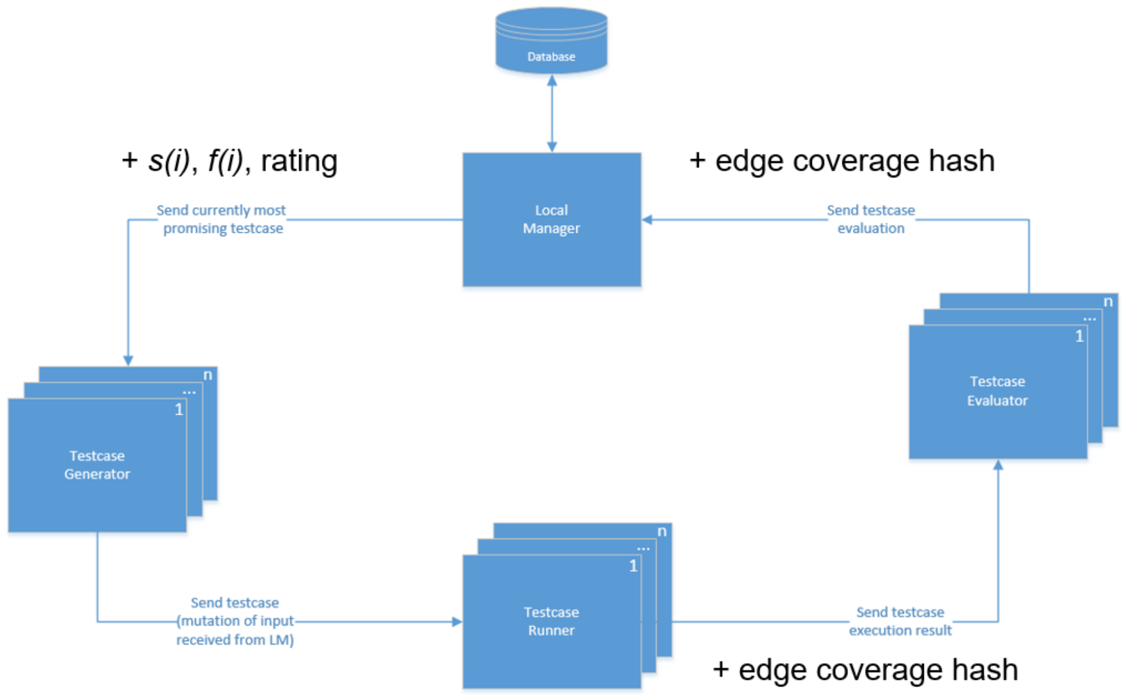

Figure 3.3: FLUFFI message changes.

message. After the Evaluator determines if the execution of the PUT discovered new basic block coverage, it sends the execution result to the Local Manager to update the database. The edge coverage hash is also added to this message. Finally, the Local Manager tells the Generator which seed to mutate. This message will now need to include $s(i)$, $f(i)$, and the FLUFFI's rating for the seed so that the Generator can compute the FAST power schedule.

3.1.4 Other Considerations

Mutator. While FLUFFI offers five different mutators, only the AFL mutator is used in the experiment. The justification is that AFLFast's experiment also used the AFL mutator.

Benchmark	Format	Seeds	Size
arrow_parquet-arrow-fuzz	Compression	1	23.6 MB
aspell_aspell_fuzzer	Text	60	4.9 MB
ffmpeg_ffmpeg_demuxer_fuzzer	Video	0	80.0 MB
matio_matio_fuzzer	Custom	6	10.7 MB
njs_njs_process_script_fuzzer	Custom	0	2.1 MB
openh264_decoder_fuzzer	Video	165	1.5 MB
poppler_pdf_fuzzer	Document	522	11.7 MB
proj4_standard_fuzzer	Custom	0	7.7 MB
stb_stbi_read_fuzzer	Image	350	143.9 KB
wireshark_fuzzshark_ip	Packet Capture	0	168.3 MB

Table 3.2: Selected FuzzBench benchmarks.

Ensuring Fairness. The edge coverage instrumentation and additional data tracking are included in all six FLUFFI implementations. First, it increases confidence that the fuzzers will operate at a similar throughput. Second, it allows for all fuzzers to be compared by edge coverage.

3.2 Experiment

10 benchmarks with known bugs are selected from FuzzBench [13], shown in Table 3.2. To ensure a diverse set of PUTs, the benchmarks are chosen such that they have diverse input types, some with and without provided seeds, and varying binary sizes to represent different levels of complexity. If FuzzBench provides no seeds for the benchmark, an empty file is used as the initial seed.

The recommendations from Klees et al. 2018 [10] influenced the parameters for the experiment. They are as follows:

- 6 implementations of FLUFFI
- 10 PUTs
- 20 trials for each implementation and PUT

- 30 CPU hour fuzzing campaign for each trial

4

Implementation

4.1 FLUFFI

All changes to FLUFFI were made from commit hash `fd9406b92f456998499703f b2d964c6e5151acea`, dated October 11th, 2021. The more recent commit did not successfully compile because of a breaking DynamoRIO update.

The majority of the work for the implementation involved adding the edge coverage instrumentation to FLUFFI. The changes include an approximately 500 line C patch for `drcov`. The shared memory is hashed using `xxHash`, a fast, non-cryptographic hashing algorithm that works at the speed limit of RAM [5]. A 64-bit hash is generated rather than AFL's 32-bits to reduce the probability of a collision. `xxHash` is chosen because it is the same hashing algorithm used in AFL++ [6].

The remaining modifications to FLUFFI are approximately 300 lines in C++, SQL schemas, and protocol buffer definitions. The SQL schema and queries are altered to support the new data being tracked. The protocol buffer definition is changed to include more information in messages passed between the worker agents.

The different search strategies are implemented by simply changing the `ORDER BY` clause in the SQL query when the Local Manager selects a seed from the queue to send to a Generator. The three search strategies can be summarized as follows:

- FLUFFI - `ORDER BY Rating DESC`
- Round-Robin - `ORDER BY TimeLastChosen ASC`
- AFLFast - `ORDER BY ChosenCounter ASC, PathCounter ASC, Rating DESC`

`PathCounter` is the counter in the edge coverage table after joining the table by the seed's hash value.

4.2 Experiment

4.2.1 Hardware and Software

The hardware and software configuration for the experiment is summarized in Figure 4.1. Four identical machines are used, each with an Intel i9-9900k CPU, 128 GB of RAM, and a 4 TB HDD. The host operating system is Ubuntu 20.04. Each host is running two LCX containers. The operating system for both containers is Ubuntu 18.04 because this version is recommended for use with FLUFFI [14]. The first container runs the Global Manager, which includes the SQL database and web interface. The second interface is running the FLUFFI worker agents. There is one Local Manager to support running one fuzzing campaign at a time. Although one Generator is often sufficient to generate enough seeds for the Runners, two Generators are running to prevent failure in the event that one dies. An equal number of Runners and Evaluators are chosen so that The Evaluators do not bottleneck runners. The hosts' CPU supports up to 16 threads, and thus the desired load average is between 15 and 16. This target load average is achieved through trial and error by provisioning 15 each of the Runners and Evaluators.

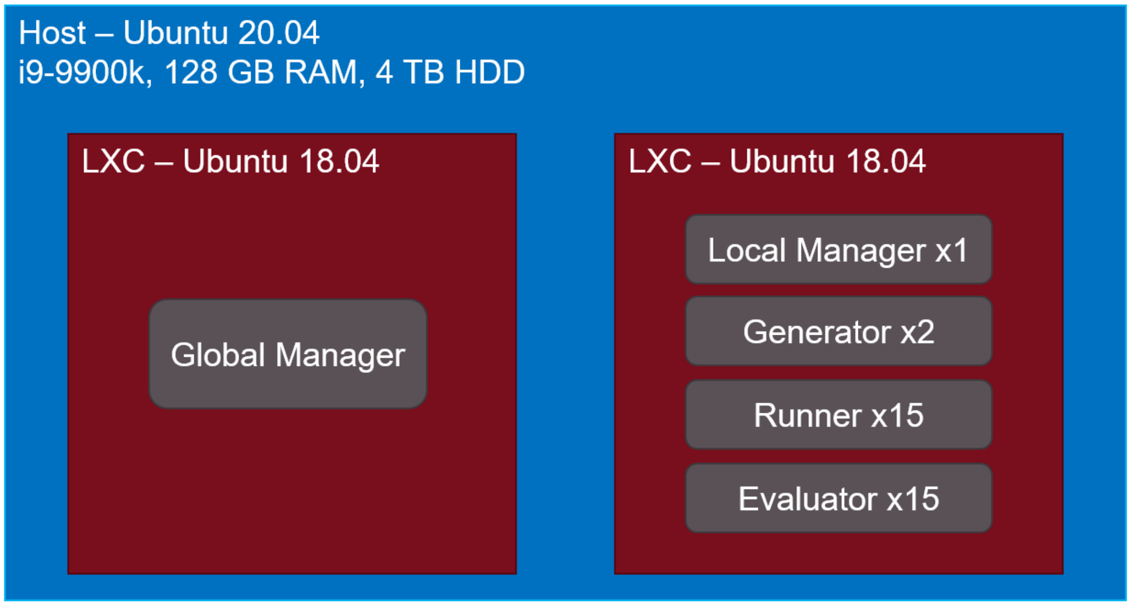

Figure 4.1: Hardware and software configuration for the experiment.

During the experiment, each of the four hosts runs a different implementation of FLUFFI. First, the four hosts run all 20 trials of the experiment for the first four implementations in Table 3.1. After this experiment completes, the final two implementations with the AFLFast search strategy are divided evenly among the four hosts such that each host completes 10 of the 20 trials for each benchmark. On all hosts, ASLR is disabled, and the performance governor is enabled.

4.2.2 Tracking CPU Time

Each trial in the experiment runs for a duration of 30 CPU hours. A CPU hour equals a process of interest scheduled to run on one CPU thread for one hour. The actual time a process runs may be longer since other processes could preempt it. While the process is waiting to be scheduled, its CPU time is not increasing. CPU time is recorded rather than real time for two reasons: (i) the experiment is run on multicore processors, which means the CPU time will increase proportionally to the number of cores being utilized; and (ii) other processes, such as another user logging in to the machine or a cronjob running, are less likely to affect the outcome of the experiment since they will not influence the CPU time of the processes of interest.

On each worker container, CPU time is tracked for the Local Manager, Generators, Runners, and Evaluators. CPU time for the children of the processes will also need to be tracked because the Runners start a new process when the PUT is executed. `/proc/[pid]/stat` provides a few relevant fields. `utime` and `stime` report the number of clock ticks scheduled for the process. `cutime` and `cstime` report the number of clock ticks the process's children have been scheduled. These values will only increase once the child process has died [2]. The `ps` command provides the `--cumulative` option that totals these four values and converts the ticks to minutes and seconds [1]. Every 10 seconds in real time, the cumulative CPU time from `ps` is summed up for each of the four different worker agents to determine the current CPU time of the experiment. PIDs and their corresponding CPU time are also tracked. If an agent restarts and its old PID no longer appears in the output of `ps`, the CPU time for the dead process is still included in the total.

4.2.3 RAM Disk

When running fuzzing campaigns with 15 Runners and Evaluators, agents would often timeout and not restart. The log files showed that SQL queries were taking up to 30 seconds. Since the Global Manager's database was running on an HDD, its I/O had become a bottleneck for the entire system. With 128 GB of RAM available, the solution was to mount `/var/lib/mysql`, which contains the files for the database, to a RAM disk. After this change, agents would no longer timeout, and the log files did not warn about slow SQL queries. The SQL database is dumped and reset after each fuzzing campaign. Thus, if the machine powered off and RAM were lost, only data for one fuzzing campaign would be lost.

To further improve fuzzing throughput, a 32 GB RAM disk is created in the worker container. The FLUFFI agent binaries and PUT binaries are copied to and executed from the RAM disk. FLUFFI's directory for seed files is also set to use the RAM disk.

4.2.4 Building FuzzBench

FuzzBench experiments are intended to be run within Docker containers. Due to the distributed nature of FLUFFI, it is not realistic to containerize the fuzzer. Therefore, the benchmarks must be built and transferred out of the Docker containers. The same linker and shared libraries for the benchmark are used to ensure the environment is as similar as possible to a traditional FuzzBench experiment. FuzzBench is forked from commit hash `2f853ecf2ae92c1284e929be31144eef924 5482e`, dated February 27th, 2022. First, all benchmark Docker containers are built. Next, a Python script is used to iterate over the containers and copy the PUT to the host. In addition to the PUT binary, `ldd` is run to determine the location of the linker and required shared libraries within the container. These binaries are also copied to the host. Finally, the initial seed files, if any, are copied to the host. When starting a fuzzing campaign in FLUFFI, the target command line is set to execute the linker with the PUT as an argument. The linker's `--library-path` option is used to set the location of the shared libraries for the PUT.

4.2.5 Experimentation Framework

Manually Managing four instances of FLUFFI is tedious. Approximately 1,200 lines of Python scripts were developed to automate various tasks. One of the scripts compiles and deploys new FLUFFI binaries to support testing of the changes to FLUFFI. Typically, a user would manually stop and start a fuzzing campaign using the Global Manager's web interface. The user would then have to wait for a periodic

Ansible playbook that occurs every five minutes to alter the agents on the worker. This process is automated by sending HTTP requests and starting the Ansible playbook manually. One of the scripts also collects data during a fuzzing campaign. Every 10 seconds in real time, the CPU time is determined. Every 10 minutes in CPU time, various statistics are collected, including the number of seeds executed, basic blocks covered, paths covered, and unique bugs found. At the end of every campaign, this data is written to an Apache Parquet file as a Pandas DataFrame. Additionally, the FLUFFI database for the fuzzing campaign is dumped and saved.

Finally, one of the scripts starts the experiment on all four machines. It automatically starts and stops fuzzing campaigns such that 20 trials are completed for each of the PUTs. Once this script was started, no user intervention was required. In 36,000 CPU hours of experimentation, only one operational failure appeared in the scripts logs. Fixing the issue required simply restarting a Docker container on the Global Manager, and the experiment picked up where it left off.

5 Evaluation

FLUFFI implementations are represented in the form of *Power Schedule / Search Strategy*. The primary goal of this thesis is to evaluate the difference in fuzzing outcomes between the AFLFast replica, *FAST / AFLFast*, and the original FLUFFI implementation, *Constant / FLUFFI*.

5.1 Average Normalized Score

Klees et al. 2018 recommend that the Mann-Whitney U test is used to compare fuzzer performance [10]. However, the statistical significance test only allows for two fuzzers to be compared at a time, and this experiment involves six different fuzzers. Furthermore, the statistical test only applies to a single PUT. A method for comparing the six implementations of FLUFFI across all 10 PUTs is desired to evaluate the experiment data effectively.

The report generated by a FuzzBench experiment includes a metric to rank all fuzzers across multiple benchmarks; average normalized score. An output variable, such as basic blocks covered, is chosen. For each benchmark and fuzzer, the median value of the output variable out of the 20 trials is determined. This median value is divided by the maximum output variable value for that benchmark out of all fuzzers. The resulting value is the normalized score for a benchmark and fuzzer. Finally, for each fuzzer, the normalized scores are averaged across all benchmarks to determine the average normalized score [13].

The average normalized score is a helpful metric for comparing all fuzzers across all benchmarks. Nevertheless, it is not a reliable figure to infer conclusions about the data. The average normalized score is used to make initial observations which can be confirmed with the Mann-Whitney U test. All instances of the Mann-Whitney U test are performed with a 95% confidence interval.

5.2 Throughput

Although the edge coverage instrumentation was added and the extra variables were tracked in the SQL database for all FLUFFI implementations, it is still possible that the throughput varies between implementations. It is essential to check if this is the case because it could explain why a fuzzer has higher performance. Fuzzer throughput is measured by the number of seeds executed during the fuzzing campaign. The number of executions is compared using the average normalized score in Figure 5.1.

The variation in throughput is reasonably low. There is only a 1.81% difference in scores between the fuzzers with the lowest and highest throughput. However, in 6 out of 10 benchmarks, *Constant / FLUFFI* has a statistically higher throughput than *FAST / AFLFast*.

Figure 5.1 shows that the AFLFast search strategy consistently has a lower throughput than the other two. The only difference in implementation between search strategies is the SQL query when selecting a seed from the queue. The query for AFLFast orders by three different columns, while the other two only reference one column. Furthermore, one of the ordered columns for AFLFast, $f(i)$, references

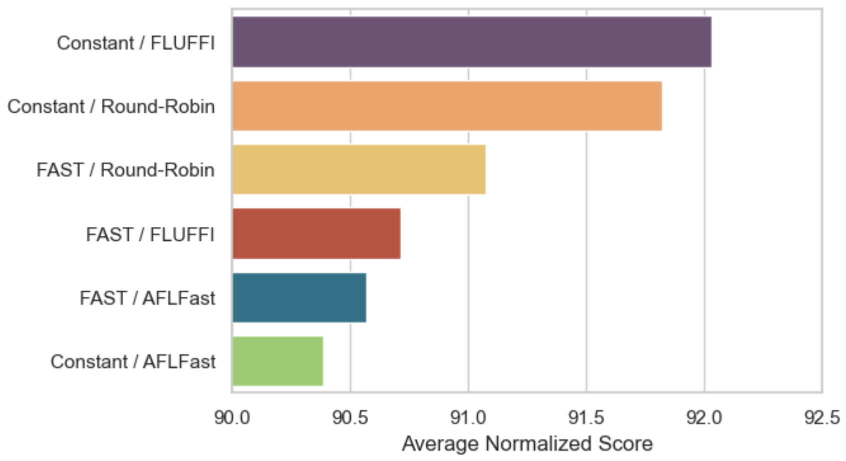

Figure 5.1: Number of seeds executed, compared by average normalized score.

a table other than the seed queue table. Thus, this query also must join the edge coverage table. The more complex SQL query is a potential explanation for the decreased throughput in the AFLFast search strategy implementations.

Another observation from Figure 5.1 is that the FAST power schedule has a lower throughput for the FLUFFI and round-robin search strategies. In implementations with the FAST power schedule, $f(i)$ must be passed from the Local Manager to the Generator. Thus, when the Local Manager queries for the next seed to be chosen from the queue, it must also select the $f(i)$ column. Doing so requires a join on the edge coverage table, potentially introducing more overhead. A possible explanation for why this is not the case for the AFLFast search strategy is that the table must still be joined for the constant power schedule.

5.3 Bug Coverage

FLUFFI reports the unique number of crashes and the unique number of access violations in a fuzzing campaign. The sum of these two values is defined as the number of bugs covered. However, FLUFFI did not discover as many bugs as

Benchmark	Maximum Bugs Found
arrow_parquet-arrow-fuzz	2
aspell_aspell_fuzzer	0
ffmpeg_ffmpeg_demuxer_fuzzer	1
matio_matio_fuzzer	33
njs_njs_process_script_fuzzer	0
openh264_decoder_fuzzer	0
poppler_pdf_fuzzer	1
proj4_standard_fuzzer	0
stb_stbi_read_fuzzer	2
wireshark_fuzzshark_ip	0

Table 5.1: Maximum unique bug coverage for each benchmark.

expected. Table 5.1 shows that no bugs were ever found for 5 of the 10 benchmarks. Only one or two were covered in most benchmarks where bugs were found. The number of unique bugs covered is compared using the average normalized score in Figure 5.2. The 5 benchmarks with no bugs covered are not included in the score.

Due to the lack of data, the variation in scores is very low. Even when comparing the highest score, *FAST / Round-Robin*, and the lowest score, *Constant / AFLFast*,

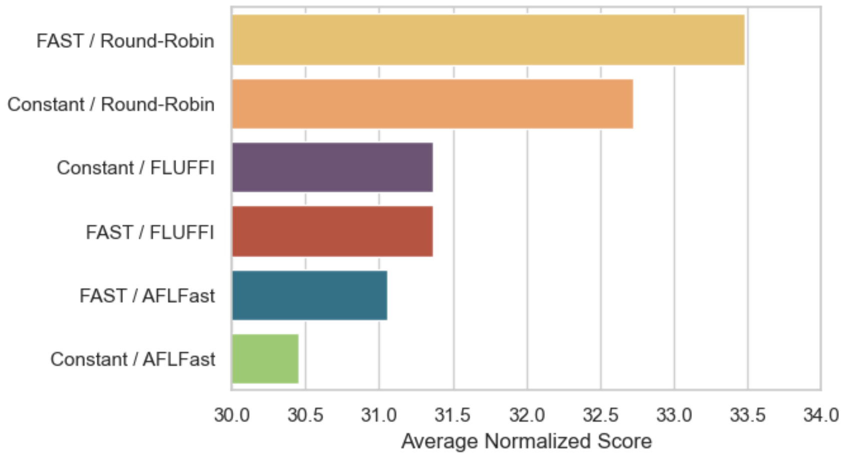

Figure 5.2: Number of unique bugs covered, compared by average normalized score.

4 out of 5 U tests are inconclusive. Thus, in this experiment, bug coverage is not a reliable indicator in comparing fuzzers.

5.4 Code Coverage

Since the edge coverage instrumentation is enabled on all FLUFFI implementations, both basic block coverage and path coverage can be used to compare fuzzers. Basic block coverage is the number of unique basic blocks covered in the PUT. Path coverage is the number of unique hashes, or paths, that the edge coverage instrumentation returned. Both values only include coverage in the PUT and not shared libraries.

5.4.1 Basic Block Coverage

The number of unique basic blocks covered is compared using the average normalized score in Figure 5.3. The top two fuzzers, both implementing the AFLFast search strategy, have very similar scores, while the rest are further behind. It seems that the AFLFast replica outperforms the original FLUFFI implementation, but this observation can be confirmed with a U test. The statistical test finds that *FAST / AFLFast* covers more basic blocks in 5 benchmarks, 3 benchmarks for *Constant / FLUFFI*, and 2 benchmarks are inconclusive.

5.4.2 Path Coverage

The number of unique paths covered is compared using the average normalized score in Figure 5.4. Like basic block coverage, the top two fuzzers are close in score, with the rest having lower scores. The AFLFast replica again has the top score, but its improvement over the original FLUFFI implementation is confirmed with a U test. The statistical test finds that *FAST / AFLFast* covers more paths in 7 benchmarks, 2 benchmarks for *Constant / FLUFFI*, and 1 benchmark is inconclusive.

The AFLFast replica outperforms the original implementation of FLUFFI in

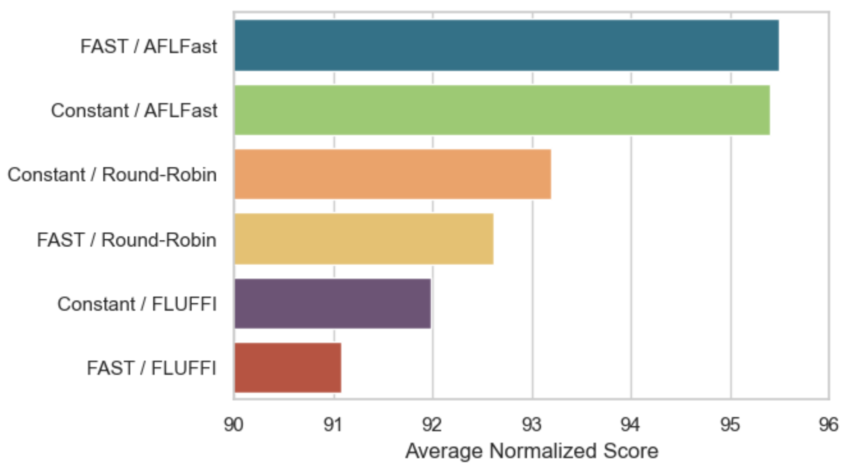

Figure 5.3: Number of basic blocks covered, compared by average normalized score.

code coverage for both metrics. However, *FAST / AFLFast* has better code coverage in fewer benchmarks if the metric is basic blocks covered. One possible explanation is that the fuzzers are "trained" to seek specific code coverage. *Constant / FLUFFI* prioritizes seeds with a higher rating, and the rating will increase when new basic block coverage is found. Thus, this fuzzer is "trained" to increase basic block cov-

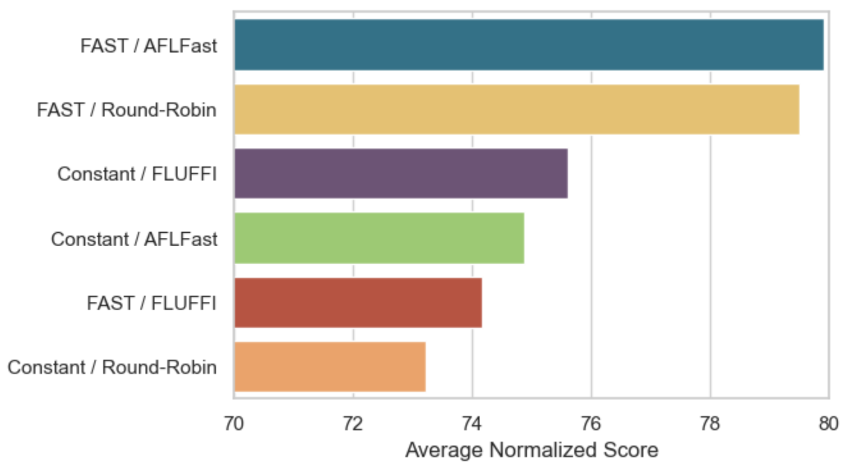

Figure 5.4: Number of paths covered, compared by average normalized score.

erage. On the other hand, *FAST / AFLFast* selects seeds with rarer path coverage. Additionally, more seeds are generated when the original seed's path coverage is rarer. Less frequented paths are more likely to find new paths. Thus, this fuzzer is "trained" to increase path coverage.

5.5 Code Coverage Relative to Throughput

The number of basic blocks and paths covered is divided by the number of seeds executed to determine the code coverage relative to the fuzzer's throughput. This approach better evaluates the algorithm's performance rather than the implementation since throughput varies slightly between fuzzers.

5.5.1 Basic Block Coverage

Figure 5.5 compares the fuzzers' basic block coverage relative to their throughput. The ranking of fuzzers is slightly different from Figure 5.3. Without comparing relative to throughput, the constant power schedule outperforms the FAST power schedule for each search strategy except for the AFLFast search strategy. When dividing by seeds executed, this relationship reverses. The FAST power schedule now has a higher score for each strategy except for the AFLFast search strategy. A potential reason is that the FAST power schedule introduces overhead by joining another table in the SQL query for choosing a seed, which may reduce throughput. This explanation would not be valid for the AFLFast search strategy because the table must be joined even with a constant power schedule.

Using a U test, the AFLFast replica can again be compared to the original FLUFFI implementation. The statistical test finds that *FAST / AFLFast* has higher basic block coverage relative to throughput in 8 benchmarks, 1 benchmark for *Constant / FLUFFI*, and 1 benchmark is inconclusive. Using the metric relative to throughput is much more conclusive in showing that AFLFast's power schedule

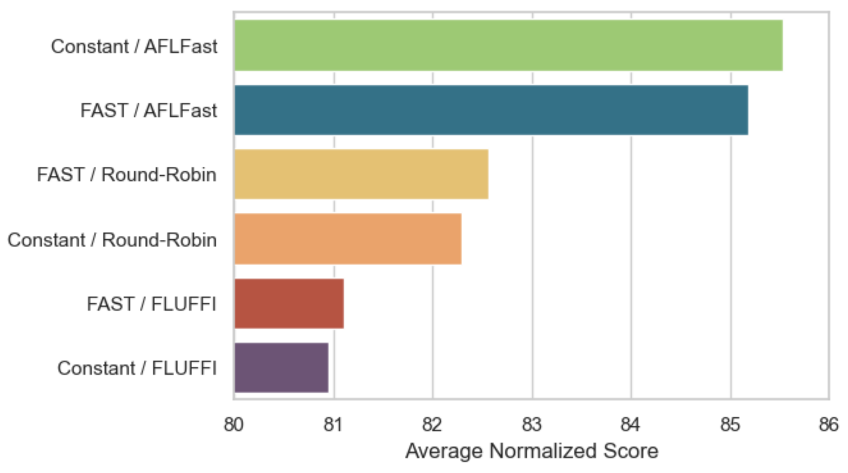

Figure 5.5: Number of basic blocks covered divided by seeds executed, compared by average normalized score.

and search strategy are the optimal algorithms for basic block coverage in FLUFFI.

5.5.2 Path Coverage

Figure 5.6 compares the fuzzers' path coverage relative to their throughput. The ranking of fuzzers is very similar to Figure 5.4, with only *Constant / AFLFast* moving ahead of *Constant / FLUFFI*. This difference may be explained because the AFLFast search strategy has more overhead in the seed selection SQL query than the FLUFFI search strategy.

Using a U test, the AFLFast replica can again be compared to the original FLUFFI implementation. The statistical test finds that *FAST / AFLFast* has higher path coverage relative to throughput in 6 benchmarks, 3 benchmarks for *Constant / FLUFFI*, and 1 benchmark is inconclusive. Using the metric relative to throughput is slightly less conclusive in showing that AFLFast's power schedule and search strategy are the optimal algorithms for path coverage in FLUFFI.

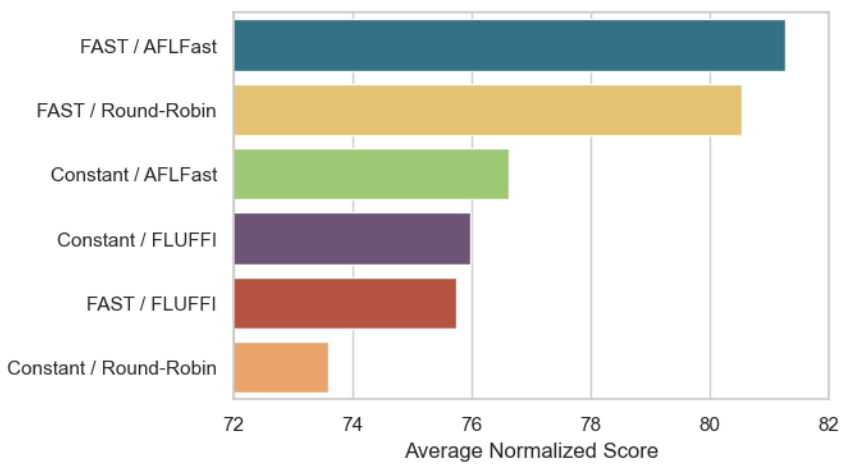

Figure 5.6: Number of paths covered divided by seeds executed, compared by average normalized score.

5.6 Code Coverage Over Time

Since measurements are collected every 10 CPU minutes, code coverage can also be compared at different times within the 30 CPU hour fuzzing campaign. The average normalized score is computed at 10 CPU minute intervals. Note that the average normalized score may decrease as CPU time increases because the maximum value is also recomputed at each time.

5.6.1 Basic Block Coverage

Figure 5.7 shows the median basic block coverage throughout the fuzzing campaign for the njs_njs_process_script_fuzzer benchmark. Coverage increases exponentially at the beginning of the campaign because a single execution can discover multiple basic blocks. As the campaign continues, the increase in coverage levels off as it becomes more challenging to cover new basic blocks in the PUT.

Figure 5.8 compares the implementations' basic block coverage throughout the fuzzing campaign. There is a more apparent separation between the FLUFFI search

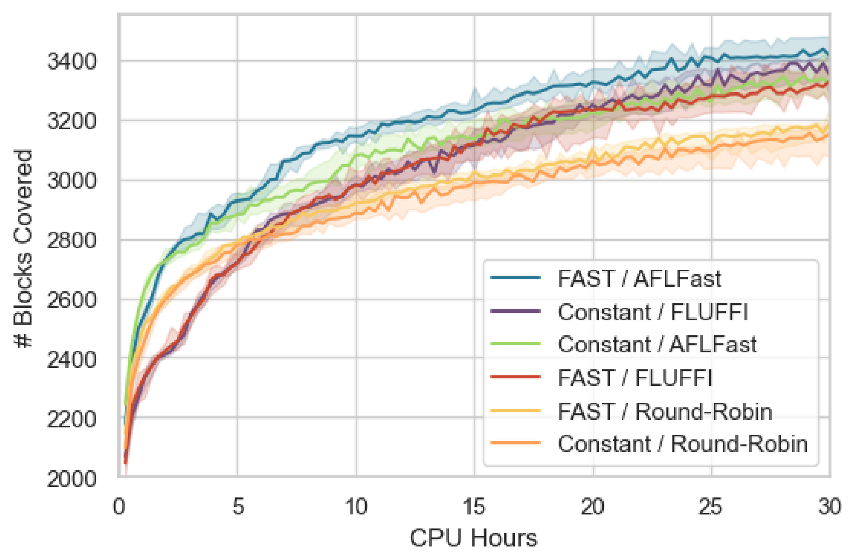

Figure 5.7: Median number of basic blocks covered over time for the njs_njs_process_script_fuzzer benchmark with 95% confidence interval.

strategy and the other two search strategies. Nevertheless, the FLUFFI search strategy implementations nearly close the gap as CPU time increases, especially with the round-robin search strategy implementations. If the fuzzing campaign had a longer duration, it is possible that the FLUFFI search strategy would match or even overtake the other search strategies in basic block coverage.

5.6.2 Path Coverage

Figure 5.9 shows the median path coverage throughout the fuzzing campaign for the njs_njs_process_script_fuzzer benchmark. Coverage does not increase as quickly at the beginning of the campaign as in Figure 5.7 because each execution can only discover one new path. New paths continue to be found at the same rate at the end of the fuzzing campaign.

Figure 5.10 compares the implementations' path coverage throughout the fuzzing campaign. One observation is that the round-robin search strategy's coverage is

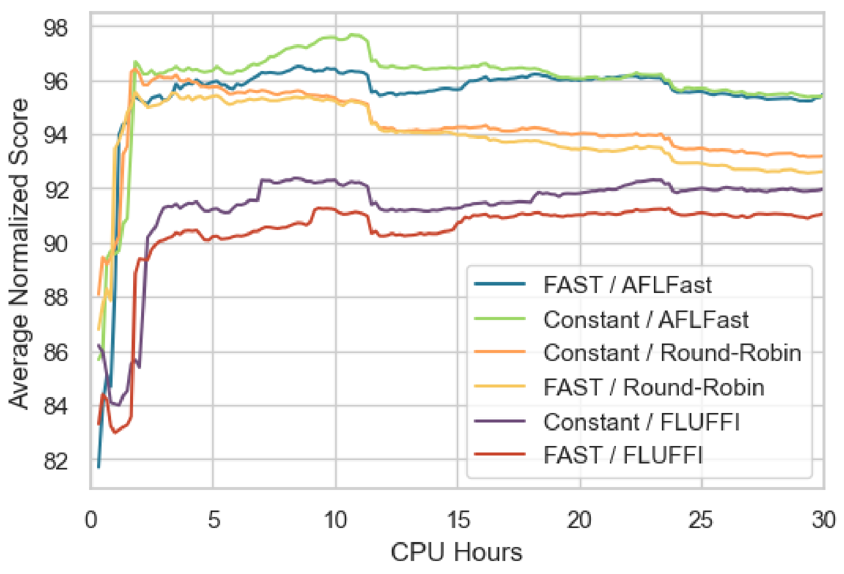

Figure 5.8: Number of basic blocks covered, compared by average normalized score over time with samples at 10 CPU minute intervals.

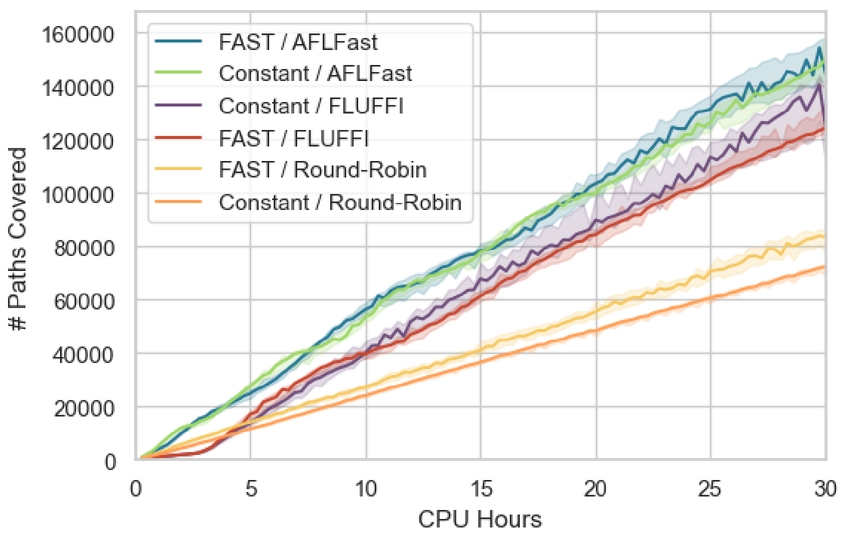

Figure 5.9: Median number of paths covered over time for the njs_njs_process_script_fuzzer benchmark with 95% confidence interval.

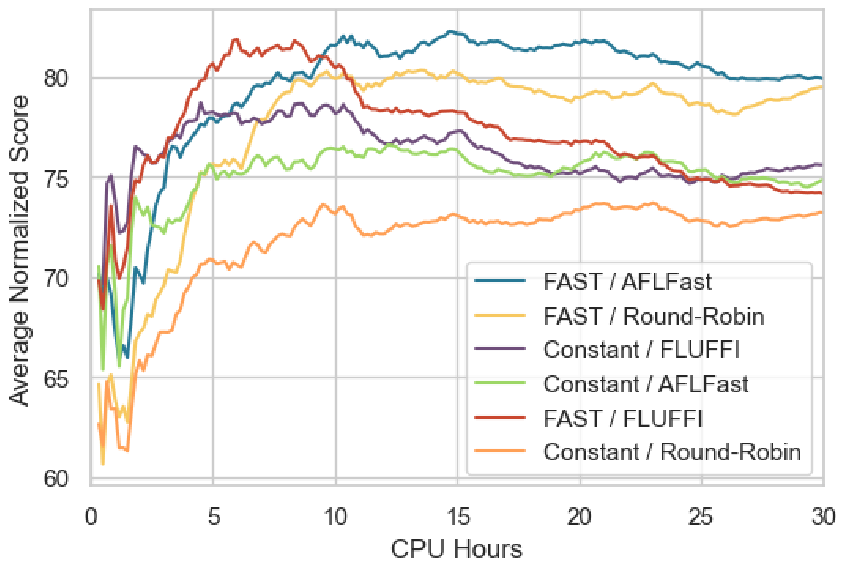

Figure 5.10: Number of paths covered, compared by average normalized score over time with samples at 10 CPU minute intervals.

much lower in the first couple of CPU hours than the other two search strategies. At the beginning of a campaign, numerous new seeds are added to the queue and are considered new. The FLUFFI and AFLFast search strategies are greedy, so they will prioritize seeds that are likely to find new coverage. On the other hand, the round-robin search strategy must fuzz all the older seeds in the queue before encountering a seed mutated from a promising previous seed, which is likely to introduce new coverage. This inefficiency potentially explains why round-robin must spend the beginning of a campaign catching up with the greedy search strategies.

Another observation is that *FAST / AFLFast* may not have the highest path coverage for the entirety of the 30 CPU hour fuzzing campaign. For example, at hour 6, it seems that *FAST / FLUFFI* has higher coverage. While a U test does not confirm this observation, statistical tests comparing *FAST / Round-Robin* to *FAST / AFLFast* at different times have interesting results. At the end of the fuzzing campaign, the U test finds that *FAST / Round-Robin* has higher path coverage in 3 benchmarks, 4 benchmarks for *FAST / AFLFast*, and 3 benchmarks are inconclusive. Thus, *FAST / AFLFast* is slightly more likely to discover more paths than its round-robin counterpart in a 30 CPU hour campaign. However, if the fuzzing campaign were 15 CPU hours, the result would be different. At this time, the U test finds that *FAST / Round-Robin* has higher path coverage in 7 benchmarks and 3 benchmarks for *FAST / AFLFast*. Therefore, *FAST / Round-Robin* would be the optimal choice for path coverage in a shorter fuzzing campaign.

6

Discussion

6.1 Related Work

Heuristics Based Mutation. Bendt shows that the FLUFFI design does not maintain some functionality for mutators since mutators are only given access to one seed to mutate rather than the entire corpus. For example, the splice operators in the AFL and HONGGFUZZ mutators combine sections of data from multiple seeds. The author proposes the OEDIPUS mutator, which merges entire seed files instead of just the head and tail. When enabling the new mutator alongside the existing FLUFFI mutators, FLUFFI can find a crash in a web server more quickly than without OEDIPUS. Bendt also proposes the CARROT mutator, which attempts to identify the length and offset fields in the input and correlate them to a string in the seed. CARROT can crash an example program with length and offset fields within seconds, whereas none of the existing FLUFFI mutators were able to find the crash within 24 hours [3].

Mutation Scheduling. Lyu et al. claim that the efficiency of mutation operators varies with the PUT and the relative time position in the fuzzing campaign. The authors introduce MOpt, an optimized mutation scheduler for fuzzers. MOpt uses a Particle Swarm Optimization algorithm to determine the optimal distribution of mutation operators dynamically. The authors' evaluation concludes that MOpt-AFL outperforms AFL, AFLFast, and VUzzer in code and bug coverage [11].

Reducing Path Collisions. Gan et al. argue that AFL's method of edge coverage results in path collisions, which prevents a fuzzer from achieving the optimal strategy. They find that, in some PUTs, 75% of edges collide with others using AFL's edge coverage. The authors propose CollAFL, which employs program analysis at instrumentation compile-time such that the probability of a collision between indexes in the shared memory array is significantly reduced. Their solution reduces the collision rate to nearly zero and can obtain higher code and bug coverage than AFL [7].

Collaborative Fuzzing. The approach of combining multiple fuzzers to improve outcomes is called *collaborative fuzzing*. Previous work, such as EnFuzz, selected combinations of fuzzers manually. Güler et al. propose Cupid, which automatically identifies sets of fuzzers that complement each other, independent of the PUT. Data for each fuzzer is collected against a suite of PUTs. Fuzzers are combined such that total coverage is maximized by combining fuzzers that solve different branches. The authors find that Cupid outperforms EnFuzz in total coverage and coverage latency [9].

Input Reduction. Another application of fuzzing is testing compilers. Inputs are generated to cover edge cases in an attempt to disclose faults in the compiler. Traditionally, inputs are generated during fuzzing and are later reduced such that they still exercise the same coverage. Vikram et al. propose *bonsai fuzzing*, which instead starts with smaller inputs and gradually increases their size until desired coverage is reached. The authors find that this technique generates 16-45% smaller inputs that still exercise the same code coverage [16].

6.2 Limitations

Mutator. FLUFFI allows for multiple different mutators to be used in a fuzzing campaign. The options are AFL, Radamsa, Honggfuzz, and two internally developed mutators. This experiment only uses the AFL mutator for fuzzing campaigns. Selecting one or more other mutators may result in different outcomes for this experiment.

Random Initial Seed. Klees et al. 2018 recommend running experiments with three different types of initial seeds; valid input, an empty file, and a random file [10]. For some benchmarks, FuzzBench provides initial seeds that are valid files. For the others, an empty file is used as the initial seed. This experiment does not include fuzzing campaigns with a random file as the initial seed.

Bug Coverage. Due to the limited number of bugs found, bug coverage is not a reliable metric to compare fuzzers for this experiment. This issue may be resolved by selecting different benchmarks or increasing the length of the fuzzing campaign.

Central Bottleneck. Despite FLUFFI being considered distributed, the Global Manager acts as a central knowledge base for the system. For example, when the SQL database's I/O was limited, it caused the database to become overloaded, and the entire system could not operate effectively.

Path Collisions. The implementation of edge coverage in FLUFFI allows for collisions between multiple absolute paths. Although `xxHash` is designed to be reasonably resistant to hash collisions, collisions are more likely to occur than for a cryptographic hashing algorithm. Path collisions could also occur because it is possible that two different edges could compute to the same index in the shared memory array. While fuzzers typically accept this trade-off for performance, the experiment could store the edge counters reported by the instrumentation rather than a hash.

Path Explosion. The implementation of edge coverage in FLUFFI is vulnerable to path explosion due to loops. Seed input may influence the number of iterations for a loop. The difference between 200 and 201 loop iterations is likely not interesting for finding vulnerabilities. The mutator could keep increasing the number of loop iterations, which will be considered a new path and be added to the queue. To solve this issue, AFL implements loop "bucketing." When AFL detects a loop, it will only keep seeds where the loop iteration count is a power of two (i.e., 1, 2, 4, 8, . . .) [15].

Consistent Coverage Metric. The original FLUFFI implementation uses basic block coverage as a metric for the search strategy, and seeds are added to the queue if they cover new basic blocks. On the other hand, the AFLFast search strategy uses the number of paths as its coverage metric, but the logic for choosing seeds to add to the queue is not changed. Therefore, if a PUT exercises a new path but does not cover any new basic blocks, the seed will not be added to the queue. This inconsistency in coverage metrics is likely detrimental to fuzzing outcomes for the AFLFast power schedule and search strategy in FLUFFI.

6.3 Future Investigations

Edge Coverage Improvements. The implementation of AFLFast's power schedule and search strategy can further be improved. First, AFL's loop "bucketing" can be added to FLUFFI to prevent path explosion due to loops. Second, the Evaluator can be changed such that seeds which cover new basic blocks *or* edges are added to the queue.

Multiple Power Schedules and Search Strategies. In the experiment, different benchmarks favor different combinations of power schedules and search strategies. FLUFFI fuzzing campaigns can be configured to set a percentage for each power schedule. When multiple Generators are deployed, they will each use one power schedule to match the distribution of power schedules. The same can be done for search strategies. When the Local Manager selects a seed from the queue, it can randomly select a search strategy from the distribution set by the user. It is possible that combining multiple power schedules and search strategies will yield better fuzzing outcomes than simply choosing one of each.

Selecting a Power Schedule or Search Strategy Based on the Campaign Length. We find that the round-robin search strategy produced higher code coverage than the AFLFast search strategy halfway through the fuzzing campaign, while the opposite was accurate at the end of the campaign. Further experimentation can be done to determine, for example, if using the round-robin search strategy for the first half and the AFLFast search strategy for the second half of the experiment results in higher coverage than only using the AFLFast search strategy for the entirety of the campaign. If a similar outcome is actual, the power schedule or search strategy can be dynamically changed in FLUFFI depending on the estimated time remaining in the fuzzing campaign.

Dynamically Changing the Mutator Distribution. The optimal mutator will depend on the PUT. For example, some mutators perform better for binary input, while others generate better text input. When users start a fuzzing campaign in FLUFFI, they can select a static distribution of five different mutators to deploy to the Generators. The FLUFFI database stores which mutator produced each seed. Therefore, FLUFFI can track which mutators are generating the seeds that find new code coverage or bugs. The distribution of mutators can be changed during the campaign to favor mutators that produce the best fuzzing outcomes.

7

Concluding Remarks

As organizations continue to adopt fuzzing as a technique to discover software vulnerabilities, they will look to distributed fuzzing to effectively utilize all of their available hardware. Current state-of-the-art fuzzers, such as AFL++, include a mode for distributed fuzzing, but it is not easy to configure, and it inefficiently syncs fuzzer states. FLUFFI is a distributed fuzzer developed by Siemens that solves these issues, but it is lacking in recent fuzzing advancements. One such improvement is AFLFast, which employs a power schedule and a different search strategy in AFL. We implement AFLFast's power schedule and search strategy, as well as a round-robin search strategy, in FLUFFI. Next, we evaluate these changes in a comprehensive experiment with 10 PUTs containing known bugs from Google's FuzzBench. Since the version of FLUFFI with AFLFast's power schedule and search strategy outperforms its original implementation in most PUTs in code coverage, we find that AFLFast's improvements over AFL can be transferred to FLUFFI. A secondary finding is that combining the FAST power schedule and round-robin in FLUFFI provides the highest path coverage for shorter fuzzing campaigns.

First, we recommend that Siemens use the AFLFast power schedule and search strategies as defaults for future fuzzing campaigns in FLUFFI. Second, we advise that more testing should be done to determine the optimal power schedule and search strategy depending on the relative time position in a FLUFFI fuzzing campaign. Finally, we recommend that AFL's loop "bucketing" and an Evaluator based on edge coverage be implemented in FLUFFI to potentially further improve the new power schedule and search strategy.

7.1 Open-Source Contributions

The relevant work for this thesis is open-sourced. It can be found in the following repositories:

- `https://github.com/sears-s/fluffi`: fork of FLUFFI with branches for the FAST power schedule and two new search strategies
- `https://github.com/sears-s/fuzzbench`: fork of FuzzBench for building all benchmarks and extracting the relevant files from the containers
- `https://github.com/sears-s/fluffi-tools`: contains scripts to manage FLUFFI, run the experiment, and analyze data, as well as the data collected during the experiment

Three pull requests are planned for `https://github.com/siemens/fluffi`. These are as follows:

- Edge coverage instrumentation and tracking
- Implementation of power schedule with option to choose power schedule in UI
- Implementation of two new search strategies with the option to choose search strategy in UI

Bibliography

[1] *ps(1) — Linux manual page*, Jun. 2020. [Online]. Available: https://man7.org/linux/man-pages/man1/ps.1.html. [Accessed 2022-02-20].

[2] *proc(5) — Linux manual page*, Aug. 2021. [Online]. Available: https://man7.org/linux/man-pages/man5/proc.5.html. [Accessed 2022-02-20].

[3] R. Bendt, "Evaluation of heuristic based input generation techniques for evolutionary binary fuzzing," Master's thesis, Hochschule München University of Applied Sciences, Munich, Germany, Jan. 2021.

[4] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing as Markov Chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 1032–1043. [Online]. Available: https://doi.org/10.1145/2976749.2978428. [Accessed 2021-10-10].

[5] Y. Collet, "xxHash: Extremely fast non-cryptographic hash algorithm," 2022. [Online]. Available: https://github.com/Cyan4973/xxHash. [Accessed 2022-02-17].

[6] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Conference on Offensive Technologies*, no. 10. USA: USENIX Association, Aug. 2020, p. 10. [Online]. Available: https://dl.acm.org/doi/10.5555/3488877.3488887. [Accessed 2021-10-15].

[7] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path Sensitive Fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*, May 2018, pp. 679–696, iSSN: 2375-1207. [Online]. Available: https://ieeexplore.ieee.org/document/8418631. [Accessed 2022-04-18].

[8] Google Inc., “WinAFL: A fork of AFL for fuzzing Windows binaries,” 2022. [Online]. Available: https://github.com/googleprojectzero/winafl. [Accessed 2022-01-13].

[9] E. Güler, P. Görz, E. Geretto, A. Jemmett, S. Österlund, H. Bos, C. Giuffrida, and T. Holz, “Cupid : Automatic Fuzzer Selection for Collaborative Fuzzing,” in *Annual Computer Security Applications Conference*, ser. ACSAC ’20. New York, NY, USA: Association for Computing Machinery, Dec. 2020, pp. 360–372. [Online]. Available: https://doi.org/10.1145/3427228.3427266. [Accessed 2021-10-08].

[10] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, “Evaluating Fuzz Testing,” *arXiv:1808.09700 [cs]*, Oct. 2018, arXiv: 1808.09700. [Online]. Available: http://arxiv.org/abs/1808.09700. [Accessed 2021-10-17].

[11] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, “MOPT: Optimized mutation scheduling for fuzzers,” in *28th USENIX Security Symposium (USENIX Security 19).* Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1949–1966. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/lyu. [Accessed 2021-10-15].

[12] V. J. M. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, “The Art, Science, and Engineering of Fuzzing: A Survey,” *arXiv:1812.00140 [cs]*, Apr. 2019, arXiv: 1812.00140 version: 4. [Online]. Available: http://arxiv.org/abs/1812.00140. [Accessed 2021-09-03].

[13] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, “FuzzBench: an open fuzzer benchmarking platform and service,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 1393–1403. [Online]. Available: https://doi.org/10.1145/3468264.3473932. [Accessed 2021-11-15].

[14] Siemens AG, “FLUFFI (Fully Localized Utility For Fuzzing Instantaneously),” 2021. [Online]. Available: https://github.com/siemens/fluffi. [Accessed 2021-11-08].

[15] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Krügel, and G. Vigna, “Driller: Augmenting Fuzzing Through Selective Symbolic Execution,” in *Proceedings of the 2016 Symposium on Network and Distributed System Security.* San

Diego, CA, USA: Internet Society, Feb. 2016. [Online]. Available: https://doi.org/10.14722/NDSS.2016.23368. [Accessed 2022-04-16].

[16] V. Vikram, R. Padhye, and K. Sen, "Growing a Test Corpus with Bonsai Fuzzing," *arXiv:2103.04388 [cs]*, Mar. 2021, arXiv: 2103.04388. [Online]. Available: http://arxiv.org/abs/2103.04388. [Accessed 2022-04-22].

Appendix A

Benchmark Basic Block Coverage Graphs

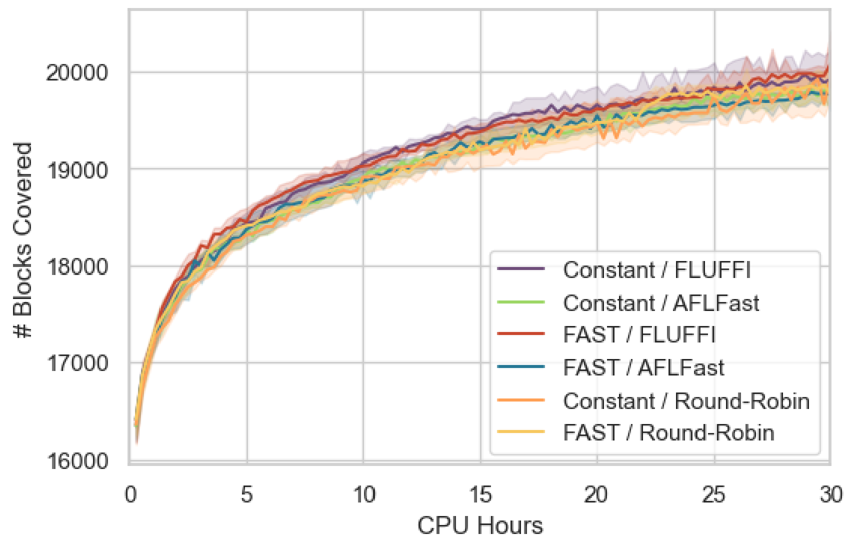

Figure A.1: Median number of basic blocks covered over time for the arrow_parquet-arrow-fuzz benchmark with 95% confidence interval.

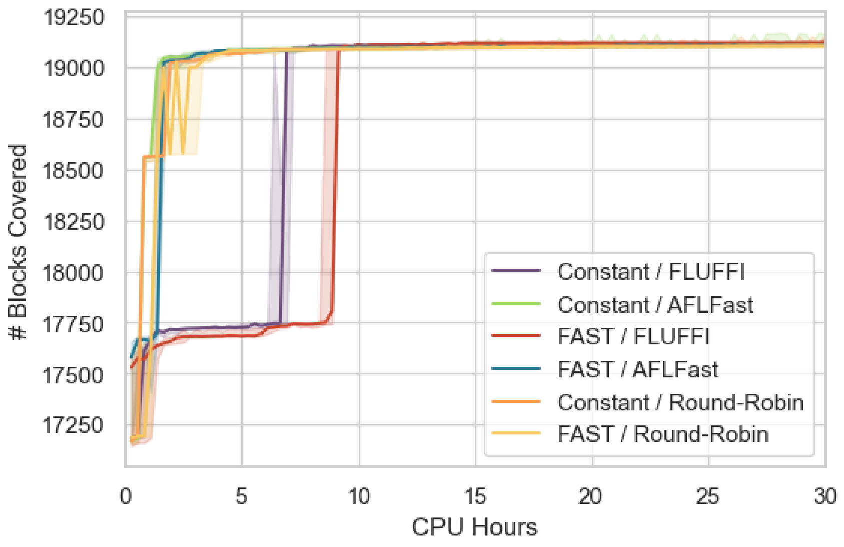

Figure A.2: Median number of basic blocks covered over time for the aspell_aspell_fuzzer benchmark with 95% confidence interval.

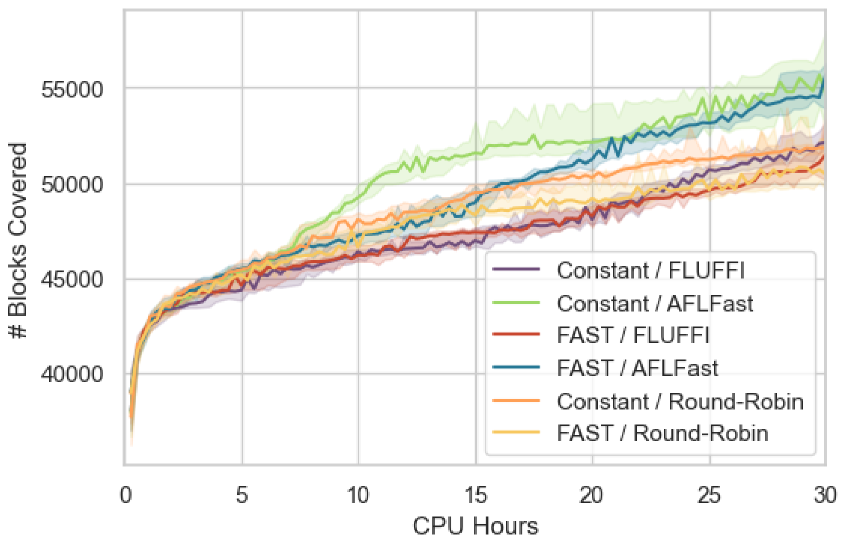

Figure A.3: Median number of basic blocks covered over time for the ffmpeg_ffmpeg_demuxer_fuzzer benchmark with 95% confidence interval.

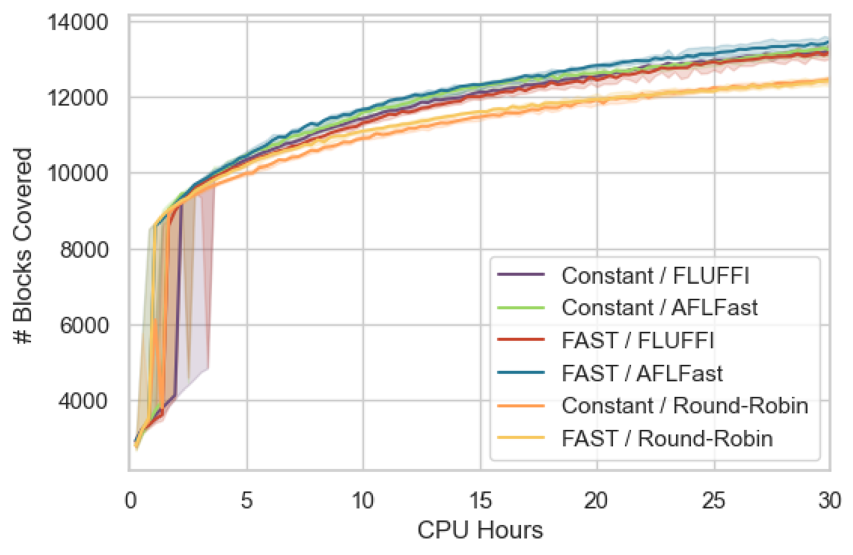

Figure A.4: Median number of basic blocks covered over time for the matio_matio_fuzzer benchmark with 95% confidence interval.

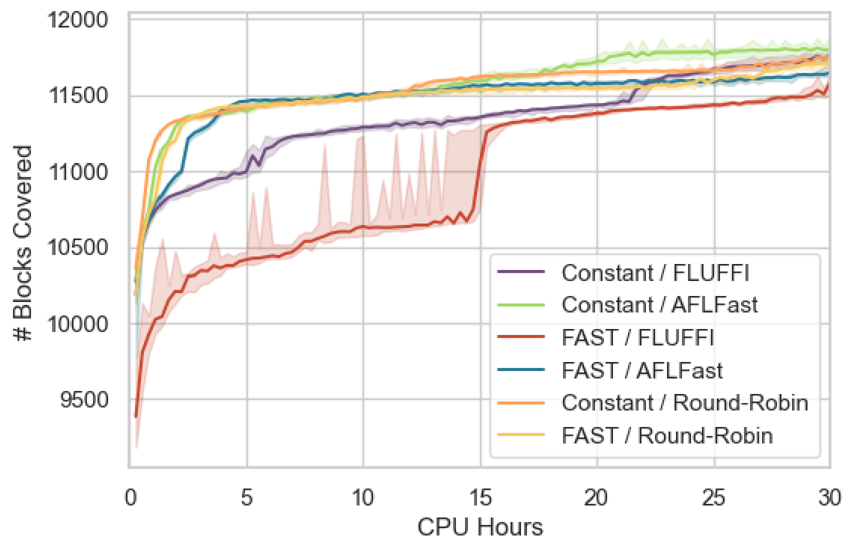

Figure A.5: Median number of basic blocks covered over time for the openh264_decoder_fuzzer benchmark with 95% confidence interval.

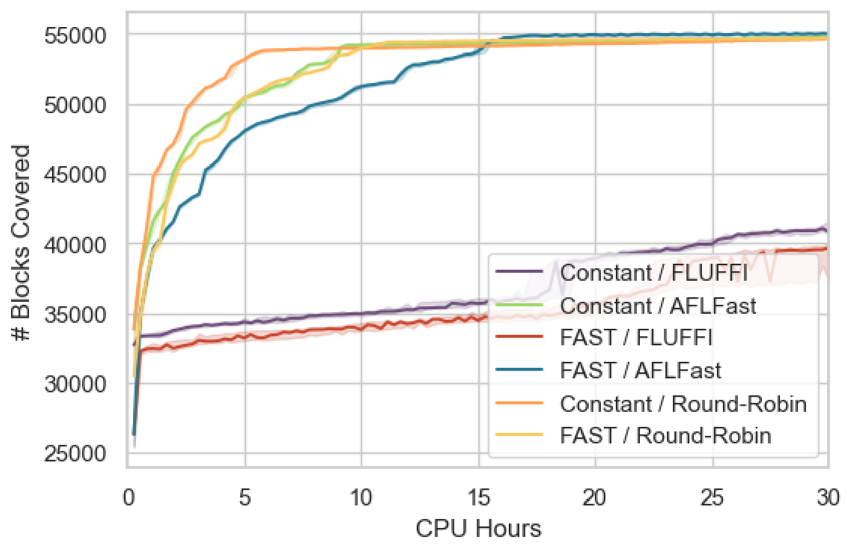

Figure A.6: Median number of basic blocks covered over time for the poppler_pdf_ fuzzer benchmark with 95% confidence interval.

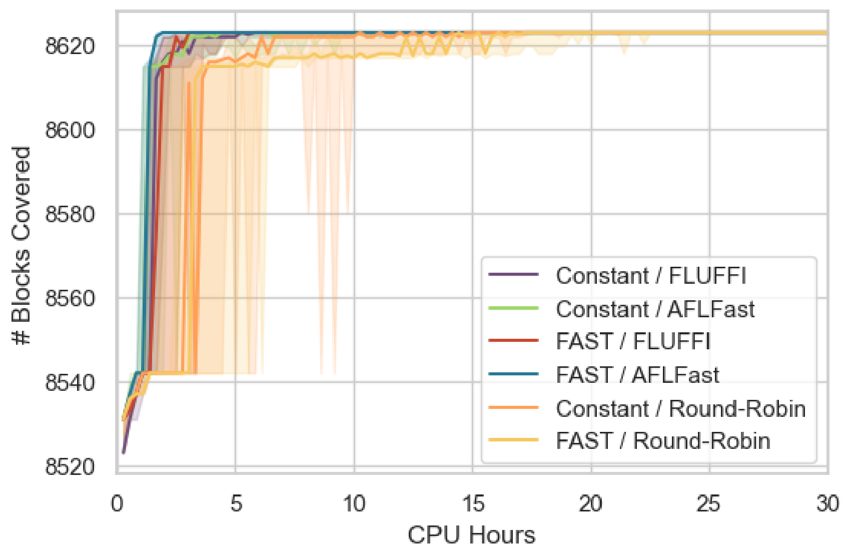

Figure A.7: Median number of basic blocks covered over time for the proj4_standard_ fuzzer benchmark with 95% confidence interval.

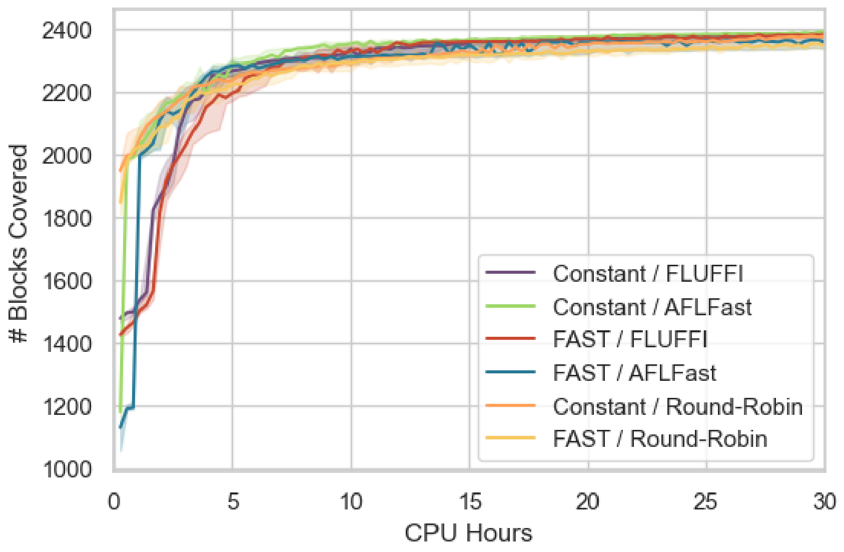

Figure A.8: Median number of basic blocks covered over time for the stb_stbi_read_fuzzer benchmark with 95% confidence interval.

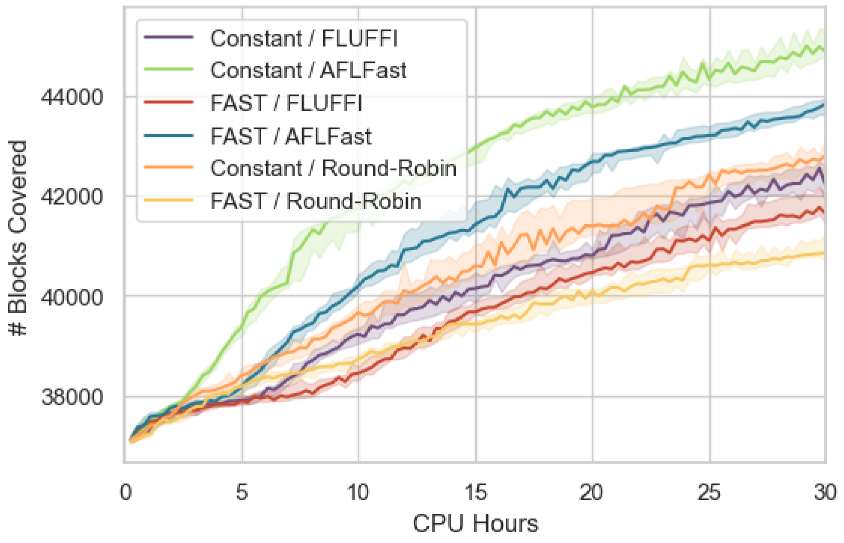

Figure A.9: Median number of basic blocks covered over time for the wireshark_fuzzshark_ip benchmark with 95% confidence interval.

Appendix B

Benchmark Path Coverage Graphs

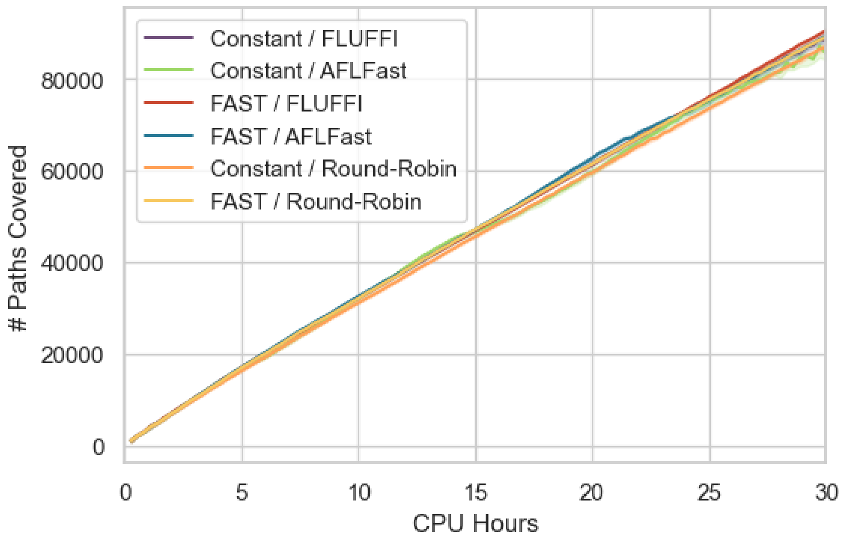

Figure B.1: Median number of paths covered over time for the arrow_parquet-arrow-fuzz benchmark with 95% confidence interval.

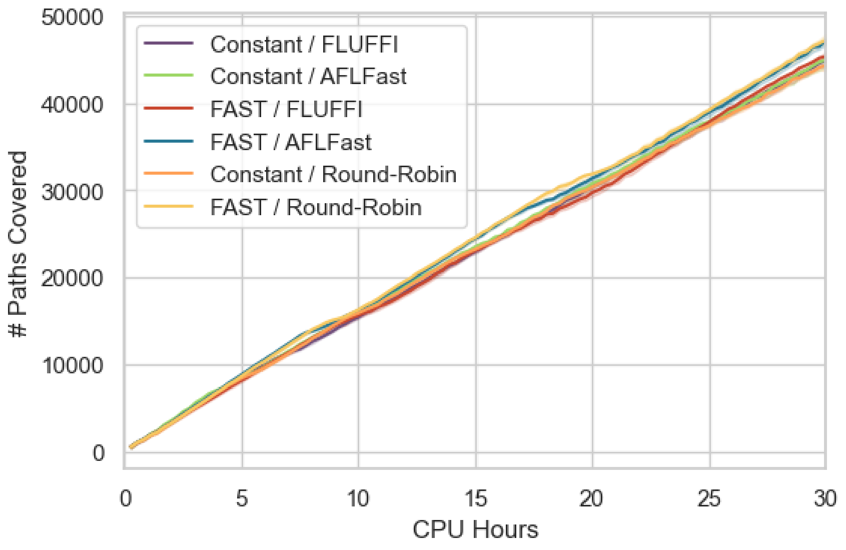

Figure B.2: Median number of paths covered over time for the aspell_aspell_fuzzer benchmark with 95% confidence interval.

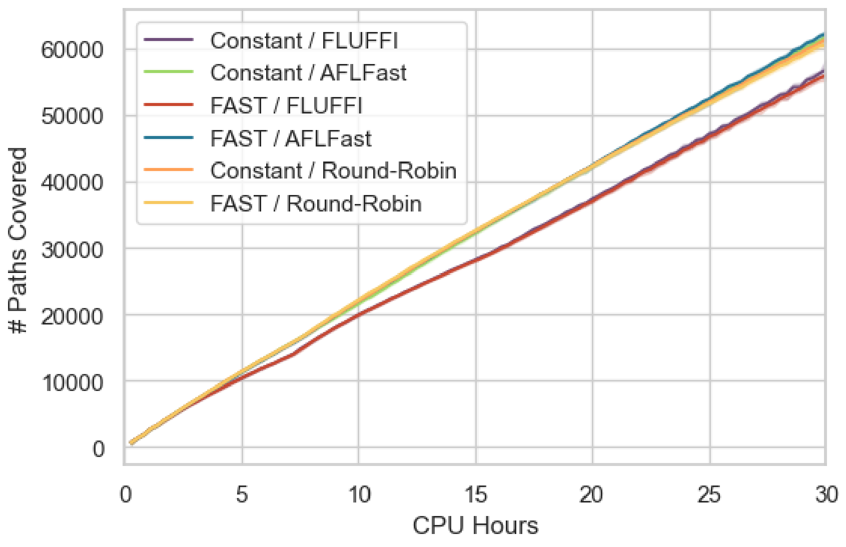

Figure B.3: Median number of paths covered over time for the ffmpeg_ffmpeg_demuxer_fuzzer benchmark with 95% confidence interval.

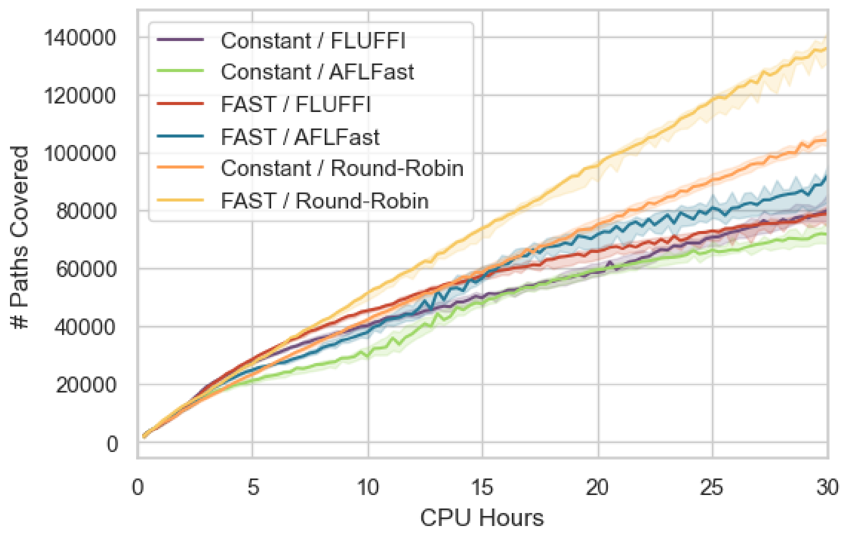

Figure B.4: Median number of paths covered over time for the matio_matio_fuzzer benchmark with 95% confidence interval.

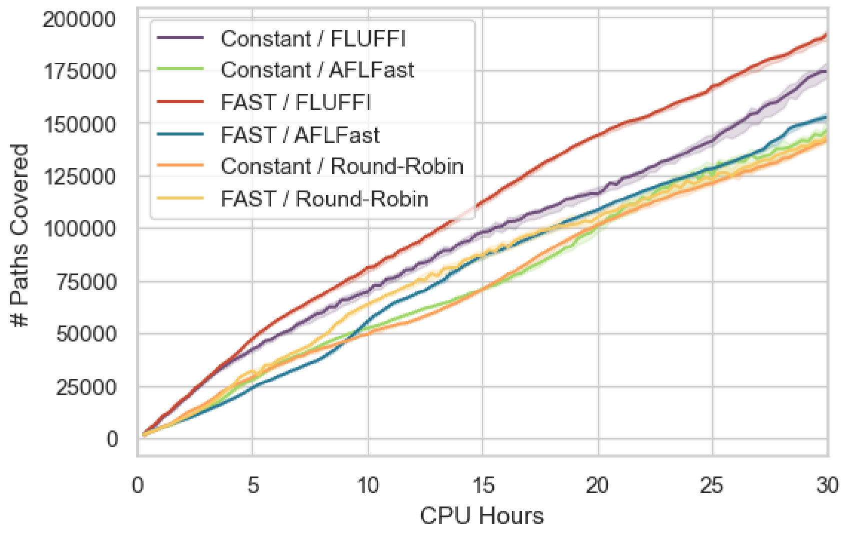

Figure B.5: Median number of paths covered over time for the openh264_decoder_fuzzer benchmark with 95% confidence interval.

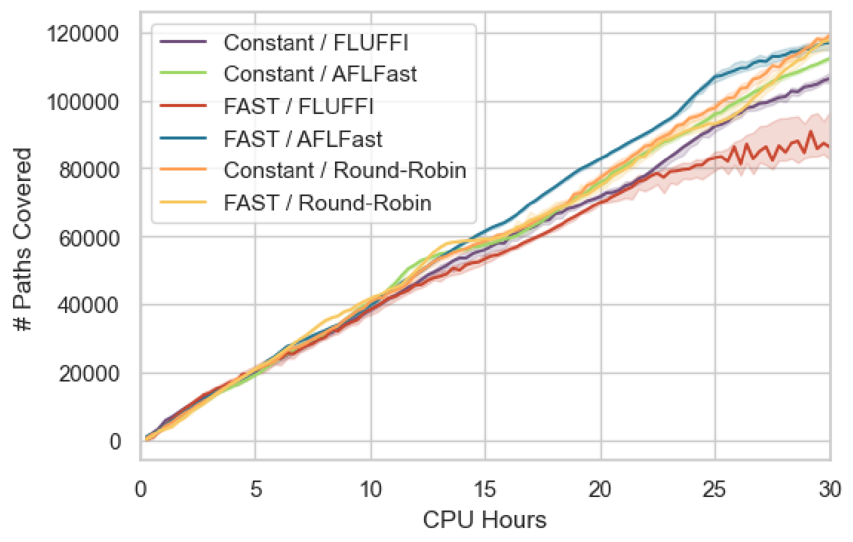

Figure B.6: Median number of paths covered over time for the poppler_pdf_fuzzer benchmark with 95% confidence interval.

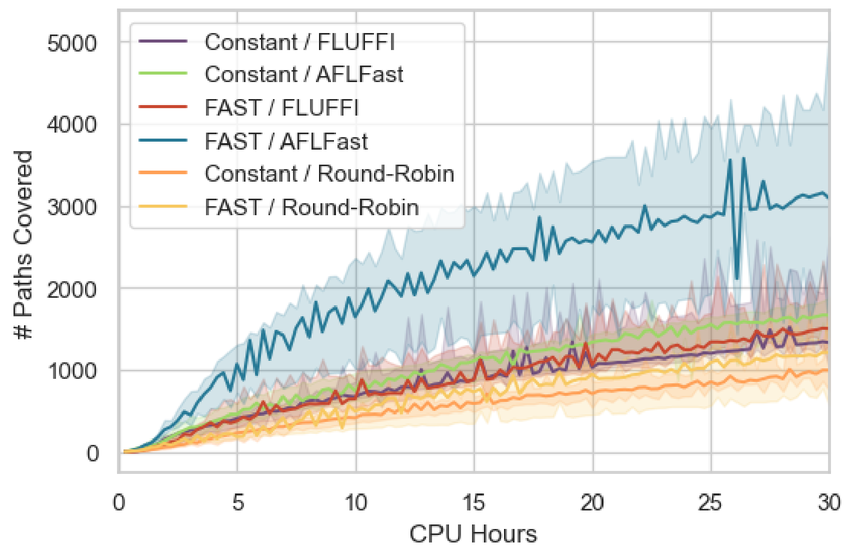

Figure B.7: Median number of paths covered over time for the proj4_standard_fuzzer benchmark with 95% confidence interval.

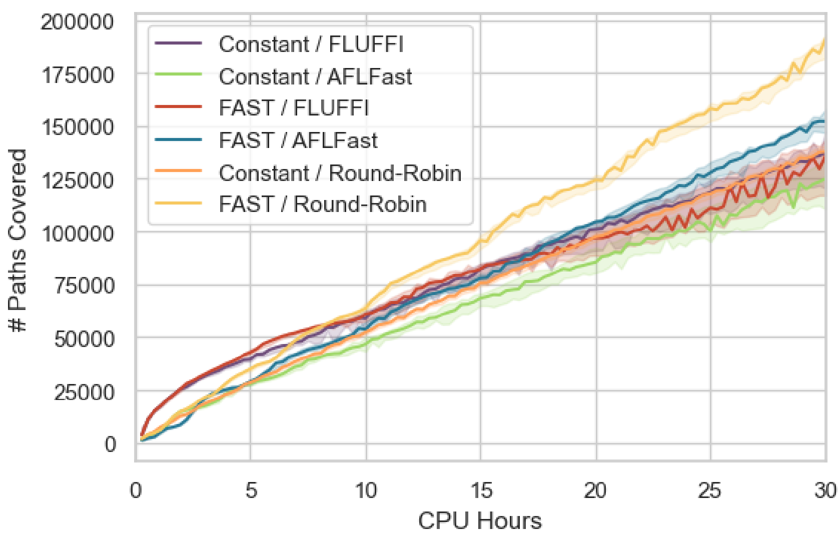

Figure B.8: Median number of paths covered over time for the stb_stbi_read_fuzzer benchmark with 95% confidence interval.

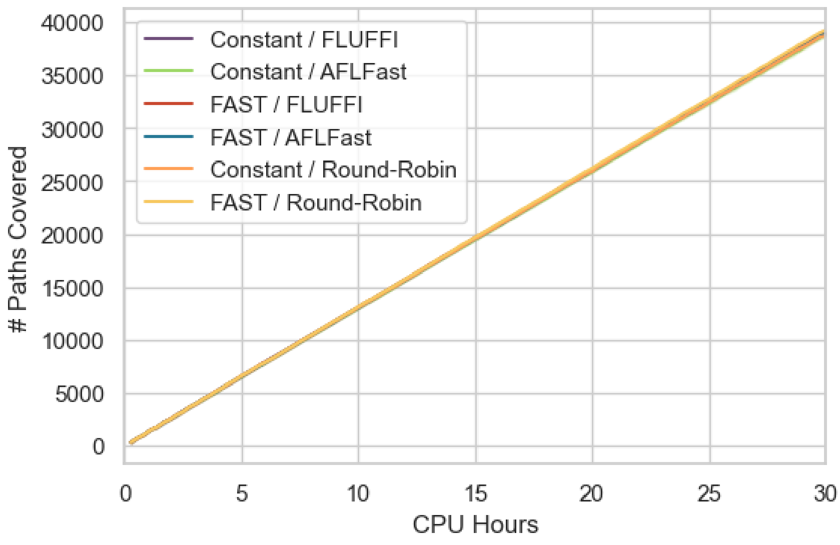

Figure B.9: Median number of paths covered over time for the wireshark_fuzzshark_ip benchmark with 95% confidence interval.