HOPPER: DISTRIBUTED FUZZER

by

Luciano Remes

cybergenik@gmail.com

A Senior Thesis Submitted to the Faculty of

The University of Utah

In Partial Fulfillment of the Requirements for the

Degree in Bachelor of Science

in

Computer Science

Approved:

Robert Ricci

Thesis Faculty Supervisor

Jim de St. Germain

Director of Undergraduate Studies

Mary Hall

Director, School of Computing

Compiled on June 16, 2023

Abstract

Software systems are becoming increasingly complex, posing significant challenges for ensuring their security and correctness. To address these challenges, researchers and industry professionals have relied on fuzzing, a technique that systematically injects random inputs into software to detect vulnerabilities and exploits. Over the past three decades, fuzzing has become a ubiquitous tool for identifying security flaws in software systems.

In recent years, there has been a growing trend towards public fuzzing campaigns, such as the OSS-Fuzz project hosted by Google. These projects have had massive impacts on the Open Source Community, driving public fuzzing campaigns for critical Open Source Software. These campaigns are typically run on the cloud, where resources can be easily scaled up or down to meet the demands of large-scale fuzzing campaigns. However, as the size and complexity of software systems continue to grow, there is a need for more efficient and scalable fuzzing techniques. We can leverage the distributed nature of the cloud to run these fuzz campaigns.

Little has been done in the space of creating truly distributed fuzzers, typically a ”parallel machine mode” is implemented in fuzzers such as AFL++. However, this is not enough and there is much still left to be done. To address this challenge, we propose a fully implemented proof of concept called Hopper, a Distributed Coverage-Guided Graybox Fuzzer that leverages the power of the cloud to distribute parallel compute across multiple nodes. Hopper aims to demonstrate a linear increase in iterations per second as the number of distributed nodes scale linearly. The inspiration for Hopper came from prior work on distributed computing techniques, such as Google's Map-Reduce, and aims to provide a proof of concept for the use of distributed fuzzing in software testing. By distributing parallel compute across multiple nodes and leveraging tried and tested techniques in distributed computing, Hopper provides a foundation for future scalable fuzzing campaigns.

Table of Contents

List of Figures

List of Tables

1 Introduction

Thesis Statement: Distributed compute applied to Fuzzing to yield a linear speedup in coverage and iterations per second.

1.1 Software & Security

With much of our societal and financial infrastructures built in software, it's become more of an incentive for bad actors or nation states to exploit vulnerabilities in Open Source systems in the pursuit of money or political cyber-warfare. These vulnerabilities not only affect nation states, but also large corporations and every-day users that rely on the correctness and safety of Open Source Software (OSS) [31]. We've begun to see large initiatives to leverage tools in the security field to find and fix these common vulnerabilities and exploits, usually called CVEs. According to a study published by Cyentia Institute and sponsored by F5 [1], "The CVE landscape has changed substantially in the last two decades, with an increasing number and widening variety of vulnerabilities" and "New vulnerability territory is being uncovered every day." It is an ever-evolving landscape, and there's still a lot that can be done to mitigate the impact of CVEs on our societal and financial infrastructure. OSS is critical as it underpins most of the modern software stacks, making them a lucrative target for hackers. One of the best known tools for finding CVEs in software is fuzzing [28, 33]. So much so, Google started an Open Source Software Fuzzing project called OSS-Fuzz [5] where they fuzz critical Open Source projects and publish vulnerabilities and exploits found for open source maintainers to fix and patch.

Typically, these kinds of large scale fuzzing campaigns are being run on the cloud. From what we've seen in the evolution of cloud computation, there appears to be a common rule: Going wide and deep have the same cost. That is to say, 10 servers for 1 hour should yield the same computation result as 1 server for 10 hours. The cloud favors distributed compute [27] and scalable infrastructures. So it's only natural that we ask, can we leverage distributed computing methods and apply them to fuzzing for large-scale fuzzing campaigns?

1.2 Fuzzing

Fuzzing has been a tool that industry and researchers have been using to find vulnerabilities and exploits in systems for the past three decades. It all started when a paper titled *An Empirical Study of the Reliability of UNIX Utilities* [17] was published in 1990, leading to a revolution in the way we think about Security. Fuzzing is a process where we take pseudo-randomly generated inputs and feed them to the *program under test* (PUT) and see where it crashes. These inputs are normally generated from a group of semantically correct beginning inputs, usually called the "corpus" of seeds. Sometimes a single input and seed are used interchangeably, and I'll be doing so in this paper.

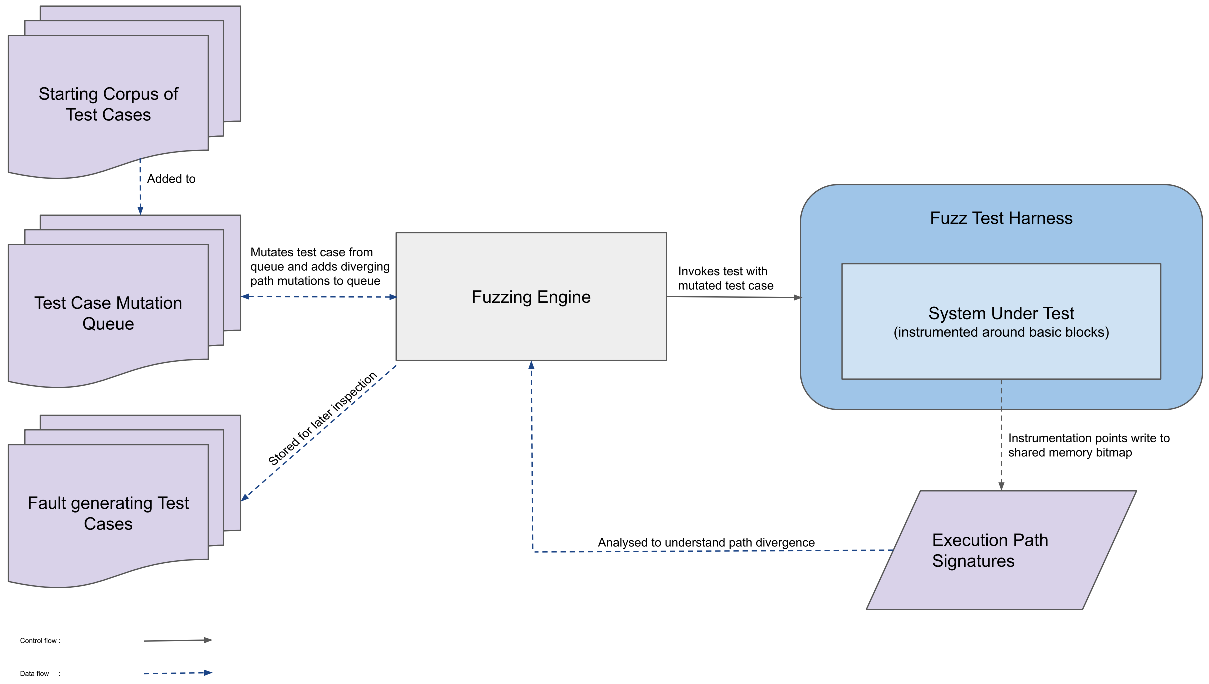

Starting Corpus of Test Cases
Added to
Test Case Mutation Queue
Mutates test case from queue and adds diverging path mutations to queue
Fuzzing Engine
Invokes test with mutated test case
Fuzz Test Harness
System Under Test
(instrumented around basic blocks)
Instrumentation points write to shared memory bitmap
Execution Path Signatures
Analysed to understand path divergence
Stored for later inspection
Fault generating Test Cases
Control flow :
Data flow :

Figure 1.1: General Modern Fuzz Architecture

In Figure 1.1 we see the general modern architecture of a mutation based fuzzer. This is taken from the AFL++ docs directly, it encapsulates well what most fuzzers do and the execution flow from a top-level perspective. In this image, the PUT is being wrapped by a *harness*. A harness is a program that is used as the entry point to the PUT, usually this is the case when the PUT is a library and so there's no main entry point to exec test inputs. Harnesses are a really common way of fuzzing targets, and can also be used to do arbitrary setup on the target before running the input. This is sometimes required as a target library might require a lot of interaction and setup before getting to the desired part of the program to fuzz. LibFuzzer is built as a modular library just for this purpose, instead of developed as a standalone executable like AFL/AFL++. This modularity allows greater flexibility, giving users more fine-grained control as to how they want to use the fuzzer.

1.2.1 Coverage

For a fuzzer to be coverage guided, the fuzzer keeps track of how much and what sections of the PUT it's inputs have covered throughout the fuzzing campaign. Coverage can be collected in multiple different ways. Some common ways are:

- block coverage: Usually this is in the form of basic blocks [4] as determined by scope of the resulting program. A basic block contains all instructions which have the property that if one of them is executed, then all the others in the same basic block are.

- edge coverage: Edge coverage uses the edges between blocks to generate a more intuitive understanding of execution flow. Consider the following graph 1.2:

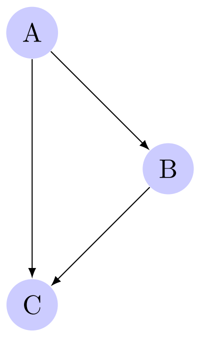
  

  Figure 1.2: Edge Graph

  If blocks A, B, and C are all covered, we know for certain that the edges A→B and B→C were executed, but we still don't know if the edge A→C was executed. Such edges of the control flow graph are called critical. The edge-level coverage simply splits all critical edges by introducing new dummy blocks and then instruments those blocks as shown in Figure 1.3. This allows us to have better understanding on coverage between edges and reduce the number of critical edges we have in the program. By simply taking them out, it disambiguates coverage.

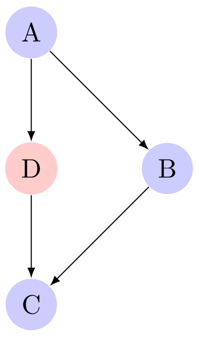


Figure 1.3: Critical Edge Graph

- line coverage: requires access to the original source code and looks at coverage on a per-line basis.

Most programs don't have this block data readily available. Most fuzzers, including AFL/AFL++ use compiler instrumentation to acquire this data. External function calls are injected at a certain point in the compilation step to send data to the fuzzer every time one of these edges or blocks is triggered. For the case of AFL++, LLVM CLANG [13] compiler was patched to inject coverage tracking as part of the compilation step. Then, a coverage bitmap is commonly used to keep track of coverage events, this is then updated upon the completion of each seed finishing it's run.

1.2.2 Mutations

Early fuzzers were primitive in trying to randomly generate inputs for a program. This is very inefficient, as usually just relying on luck to find your way to an input that will not be immediately rejected by the PUT is not a fruitful approach. Modern fuzzers employ a mutation-based fuzzing technique, as mentioned before, where they take an initial corpus of seeds. These seeds are expected to be semantically correct and valid inputs that the program will not error out on. Generating interesting inputs from a valid and semantically correct corpus yields a much higher success rate.

Mutation algorithms vary widely, some use stochastic modeling to try to do mutations on specific parts of the input that have historically shown success. Mutation algorithms are either deterministic or non-deterministic, but most will include some kind of randomness. In the case of deterministic mutation algorithms, it's common to seed a random number generator. AFL++ and AFL-like fuzzers, usually tend to be simple and do simple byte/bit-level mutations on the input. Such as swapping bytes, deleting bytes, adding bytes. This usually provides a slow enough change in the seed population such that it'll escape local minimums and not miss interesting inputs.

1.2.3 Visibility

Visibility refers to the amount of information that is exposed to the runtime of the fuzzer, on the PUT.

- Blackbox: The fuzzer has no access to the source code and does not rely on visibility in the form of symbols or sections in the target binary.
- Graybox: The fuzzer has some visibility on the original source code, but through the form of symbols and special compiler instrumentation that was done in the beginning of the fuzzing campaign.
- Whitebox: The fuzzer has full visibility into the source code of the program, giving them full information on logic and structure.

1.3 Hopper

Hopper was conceptualized in September 2022, development began almost immediately, with lots of architectural revisions along the way. Hopper's goal is an applied exploration of different distributed systems concepts and fuzzing. It's heavily inspired by AFL++ and what's been born out from the literature on Coverage-Guided, Mutation, Graybox fuzzing. It does not aim to replace AFL++ in most instances, but to merely provide a foundation for how we can build distributed fuzzing in the future. Focusing heavily on trying to build architecturally sound infrastructure as the number one priority. With optimizations in seed scheduling, mutation engine, and energy calculation falling to a distant second. Careful thought was placed into trying to build a well-rounded foundation that would provide a usable fuzzer, but most of the effort in the short time was placed on the distributed architecture.

Hopper uses a Master-Node-style as it's distributed compute schema. Hopper is built in Golang due to Go's first-class support for concurrency and parallelism, with cheap go-threads, and it's proven track record of handling large scale systems made it an obvious choice. Hopper is composable in that it's built as a library first, much like LibFuzzer [20], it can be used as a stand-alone program or be used in parts, separated into Master and Nodes. Inspired by state-of-the-art systems like AFL++, which have proven that Coverage-Guided Graybox Fuzzing is a very efficient strategy [18]. Hopper uses edge-tracking as it's coverage guidance and uses compiler instrumentation to produce a Graybox binary, similar to AFL++.

Hopper leverages techniques in distributed compute to distribute fuzzing workload across a distributed environment. Naturally, it incurs a level of latency from the RPC calls and IO operations required for communication between the nodes and the master. But it gains in that it can handle thousands of nodes in parallel, fuzzing the same target. As Hopper provides ad hoc scalability and modularity, in lieu of a more primitive *multi-machine mode* in AFL++ and other non-distributed fuzzers. I evaluate Hopper against common GNU [3] binary utilities such as *readelf*, *objdump*, and *strings*. As well as providing a series of build utilities for orchestrating Hopper and building containerized fuzzing campaigns. Hopper is currently available for use and evaluation on GitHub.

2 Background

2.1 Overview

In this section, I'll provide a background to several topics related to the design and implementation of Hopper. These being distributed computing, distributed fuzzing, and several popular fuzzers and their approaches to distributing fuzz tasks. (AFL++, Centipede, UltraFuzz). I'll provide the context for why the current scene necessitates the creation of Hopper, a new distributed fuzzing architecture.

2.2 Distributed Computing/Fuzzing

Distributed computing is a field of Computer Science that studies distributed systems. A distributed system is a system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another from any system. The components interact with one another in order to achieve a common goal.

When Google released the original MapReduce paper in 2004 *MapReduce: Simplified Data Processing on Large Clusters* [2], it left profound ripples in the Distributed Systems community. Influencing future distributed compute techniques like Apache Spark, now an industry standard. It was a great solution to the ever-growing distributed compute problem that cloud providers were starting to face, and ever since then new technologies that power cloud compute like Apache Spark have been made atop the shoulders of MapReduce. These 2 projects have inspired the work with Hopper.

State of the art fuzzers like AFL++ implement primitive, so called, multi-machine modes. These multi-machine modes are essentially just sshing into different machines and running a copy of AFL, then having a sync directory. Then occasionally syncing the directory of seeds. This wouldn't really be considered a distributed system, in the modern sense. While there is information being shared between nodes, there is no sense of direction or explicit coordination. It just syncs the seeds on an arbitrary interval. In fact, AFL++'s only documentation on running on multiple machines is a subsection of a page where they talk about how to sync nodes with shell scripts in their docs.

2.3 AFL++

AFL++ is by far the most popular fuzzer, it's a Coverage-Guided Mutation-based Graybox Fuzzer. It is the spiritual and predominate successor of the original AFL project, which is now the industry standard and has become ubiquitous within the Security and Fuzzing field. It's written in C++ and is open source, giving it a great advantage in receiving open source contributions. It's being openly developed on Github [7], with over 6,000 commits and over 150 contributors. AFL++ is one of the most contributed open source projects and an essential security tool in the open source community.

The design of AFL++ also employs a subset of genetic algorithms for doing mutations. Mutated test cases that produce new state transitions within the program are added to the input queue and used as a starting point for future rounds of fuzzing. They supplement, but do not automatically replace, existing finds. The current number of seeds that are ready to be fuzzed can be considered the population. The objective is to move that population, individual by individual, towards higher coverage. AFL++ heavily relies on compiler instrumentation and pruning of the initial corpus, through different adjacent programs that attempt to optimize initial seed count and coverage of each seed. By pre-executing and running some heuristics on a corpus, these tools can trim down the corpus into smaller and more significant inputs.

AFL++ is what's called an *out of process* fuzzer, meaning that to execute the PUT, AFL++ has to do a fork-exec command. This is usually considered quite expensive since the program has to copy its entire program state when forking. A different method, uses in-process fuzzing like in the case of LibFuzzer. A program that wraps the target program and can be linked against the fuzzer, usually referred to as a harness. Such that the fuzzer need only make a function call to test a seed. AFL++ uses a design paradigm to continuously fork itself called a *fork-server*, where part of the program's only purpose is to fork off upon each execution. The following is a general outline of the steps AFL++ takes, as found on GitHub [8]:

1. Load user-supplied initial test cases into the queue.
2. Take the next input file from the queue.
3. Attempt to trim the test case to the smallest size that doesn't alter the measured behavior of the program.
4. Repeatedly mutate the file using a balanced and well-researched variety of traditional fuzzing strategies.
5. If any of the generated mutations resulted in a new state transition recorded by the instrumentation, add mutated output as a new entry in the queue.
6. Go to 2.

AFL++ has also inspired a diaspora of AFL-like fuzzers that have emerged, making small adjustments to different techniques, all seeking to improve performance on some metric. This has been the common trend in fuzzer related papers, alter AFL++ in some meaningful way, and publish findings based on these minor changes.

2.4 Centipede

Centipede [22] is a distributed fuzzer being developed by Google, written in C++ [23]. It was open sourced on July 2022. As previously mentioned, the original authors of LibFuzzer saw that AFL and AFL-like fuzzers were not pushing towards a distributed architecture. Which is the direction that the industry is going towards, significant fuzzing campaigns are almost exclusively being run by large corporations like Google with OSS-Fuzz. As fuzzing is a computationally intensive task, to hit the plateau of the logarithmic curve can take anywhere between 12 and 24+ hours. Then coverage begins to drop substantially, but as software is being constantly developed, there's a constant need to reach that plateau upon every new release. How best to utilize a distributed set of resources than to use tried and tested methods in distributed computing. They felt this so strongly that they began working on a completely new project, centipede, from scratch.

Centipede will be a formidable competitor in the future of distributed fuzzing. However, one of the largest issues is the age-old problem of creating correct distributed systems with a language like C++. Precisely why Google and Rob Pike began development on Golang [10] for his despise of C++ safety issues, and hard to use nature. Golang has now become a major player in developing large, scalable infrastructure, and has stood the test of time. The future of distributed fuzzers is likely to be in Rust [11] or Golang, languages that offer better memory safety guarantees and ease of use.

2.5 UltraFuzz

UltraFuzz [32] is the most recent work published in the area of distributed fuzzers. It uses a 2017 version of AFL as it's main fuzzing engine. Although no diff was provided. While AFL is still a great fuzzer, it lacks a lot of modern upgrades that have been brought in AFL++. UltraFuzz breaks down it's fuzzing into 3 different instance types:

- Evaluation instances: Spawned threads to handle seed evaluation.
- Fuzzing instances (AFL): AFL instances that request seed from MongoDB, run it, and put results back into mongo.
- Main Node: Runs scheduler, coordinates spawning of evaluation instances, and assigns tasks to fuzzing

instances.

All distribution of seeds and communication of coverage results is done through the MongoDB instance. The Main node will assign a task ID over RPC which is associated to a seed in Mongo. MongoDB is then used as a key-value store to handle concurrent communication between the main node and fuzzing instances. Each fuzzing instance keeps a bitmap of their own coverage, they periodically synchronize bitmaps with a global bitmap held by the Master, which is then propagated to fuzzing nodes on a set interval. The seed selection and scheduling algorithm used in UltraFuzz, is essentially the same as AFL, except for some minor changes to distribute workload. UltraFuzz incurs the cost of running a MongoDB instance in lieu of a well-designed communication schema that could be carried out by the Master. The use of MongoDB in this instance, while very clever as it that avoids a lot of architectural pitfalls, is wasteful in resource usage.

UltraFuzz was an exceptionally difficult fuzzer to use, as very limited documentation is provided on running it. After waiting 10 seconds for Gitlink, the Chinese version of GitHub to load their published repo, we kept on having issues with getting UltraFuzz to compile, let alone run. Also, some of the documentation on using *pfuzz_web* was in Chinese, so this made it a bit difficult to work with. Due to these issues, we have not been able to run a full campaign in UltraFuzz.

2.6 Why Hopper?

Little has been done to investigate distributed fuzzing, with only 2 notable works having come out in the past year. More recently, a paper was released proposing a new distributed fuzzing model called UltraFuzz in November 2022. Google open sourced Centipede in July 2022, there was little work that had been published before that in the space. Parallel/Multi-machine modes provided by AFL++ and LibFuzzer are not leveraging all that distributed compute has to offer, there are smarter ways of distributing fuzzing campaigns. The clear scaling limitations of AFL++ and LibFuzzer in parallel/multi-machine mode environments, necessitates an urgent investigation into the use of distributed fuzzing. Where critical state can be shared among clusters of machines, work can be intelligently allocated, and thus, reduce the amount of mutation and coverage collisions. It's critical to minimize repeated work, specially in a distributed environment with larger seed sizes that incur a large cost of running. It seems imperative that more research be done in this space. A concern that the original creators of LibFuzzer shared publically on the LibFuzzer website:

"The original authors of libFuzzer have stopped active work on [LibFuzzer] and switched to working on another fuzzing engine, Centipede." [21]

"We are trying to build Centipede based on our experience with libFuzzer (and to less extent, with AFL and others). We keep what worked well, change what didn't." [22]

With this in mind, and a serendipitously timed assignment to implement Map-Reduce in a Distributed Systems course taught by Professor Ryan Stutsman. The design and development of Hopper began. As stated previously, Hopper does not aim to replace AFL++ in most instances, it's an exploration into a seemingly emergent field of distributed fuzzing. With hopes of providing a foundation for how we can build distributed fuzzers in the future.

3 Design

3.1 Overview

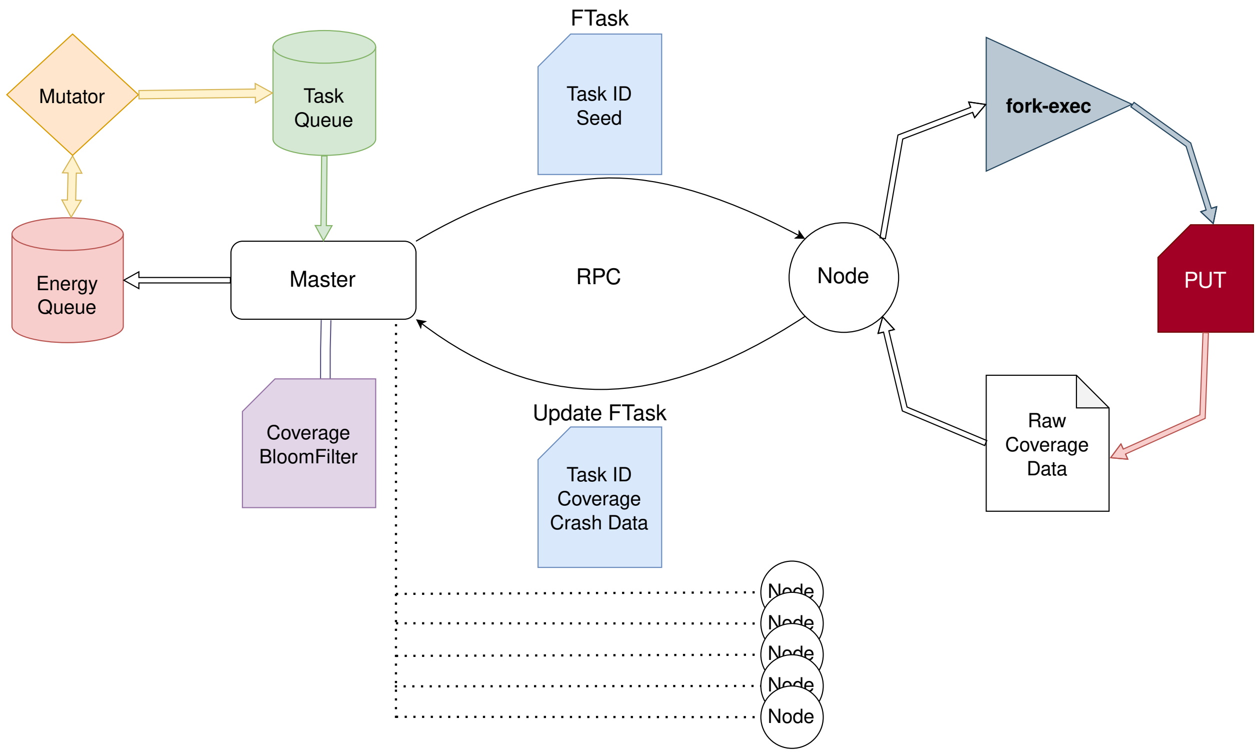


Figure 3.1: Hopper Architecture

Hopper is a modular, distributed, coverage-guided, graybox fuzzer inspired by AFL++, the current state of the art fuzzer. AFL++ inspired its mutation algorithm, energy assigning strategy, and out-of-process coverage gathering. For scheduling and several synchronization tasks, Hopper follows a Master-Minion/Server-Client architecture. There is only one Hopper Master and there are many Hopper Nodes. These nodes communicate over RPC with the Master to receive fuzzing tasks and consolidate coverage data. The nodes do not communicate amongst themselves, and so any communication about the cluster is only accessible through the Master. That is to say, for observability, one needs to have access to the

Master. Hopper is a standalone, new piece of software developed from scratch, implemented in Golang. The system has been designed as a modular library, the Master and Nodes are decoupled modules, allowing for extendability and better usability. This allows users of Hopper to build software on top and around the core Hopper components, giving us a more customizable system. It also makes it trivial to make distribution scripts for campaign setup and controlling campaign resources.

3.2 RPC Schema

The communication schema between the Master and all the Nodes, relies on a task ID. This is called the "FTaskID", and it's a hash identifier used by the Master to keep track of fuzzing tasks, progress, and coverage information. A FTask is a fuzzing task that contains an ID and an input. For the sake of communication, a seed will always have an attached FTaskID to it. The FTaskID is shared in all RPC calls from the Master and Node. There are 2 general types of RPC calls between the Master and Node:

- Get FTask: This type of RPC call is from a fuzz Node to the Master. The Master assigns a FTaskID to a seed and replies with the seed and the FTaskID.
- Update FTask: This type of RPC call is from a fuzz Node to the Master. After finishing a fuzz job, the Node sends an update for that particular task, and includes critical information: FTaskID, coverage, and crash data.

3.3 Master

The Master node's job is to schedule fuzz tasks on Nodes in the cluster, keep track of coverage, mutate seeds, and produce reports. The Master handles all these responsibilities concurrently [25]. There are two main processes running concurrently on the Master, an RPC server and the Mutation Engine. When receiving an update for an FTask, the Master uses an energy mutation algorithm to rank a seed's value, this is referred to as the energy of that input. We attach this energy to the seed, that energy is used as the rank ordinal in the energy priority queue (EPQ). Seeds for which we've done this calculation for are called energized seeds.

The mutation engine acts as a load balancer by popping energized seeds from the energy priority queue, mutating them, and feeding newly formed seeds to the task queue, as can be seen in Figure 3.2. The Mutation Engine only mutates when there's enough space in the Task Queue for more tasks, otherwise it stalls. Because a single energized seed can turn into tens of seeds, this can be seen as an inverse funnel, thus the Mutation Engine has some control of *flow* through the system. The Master, while concurrent, is mostly event-driven. As seen on Figure 3.2, most of the operations on the Master are influenced or triggered by an

RPC call. A step by step breakdown of the flow of a single fuzz task through the Master is as follows:

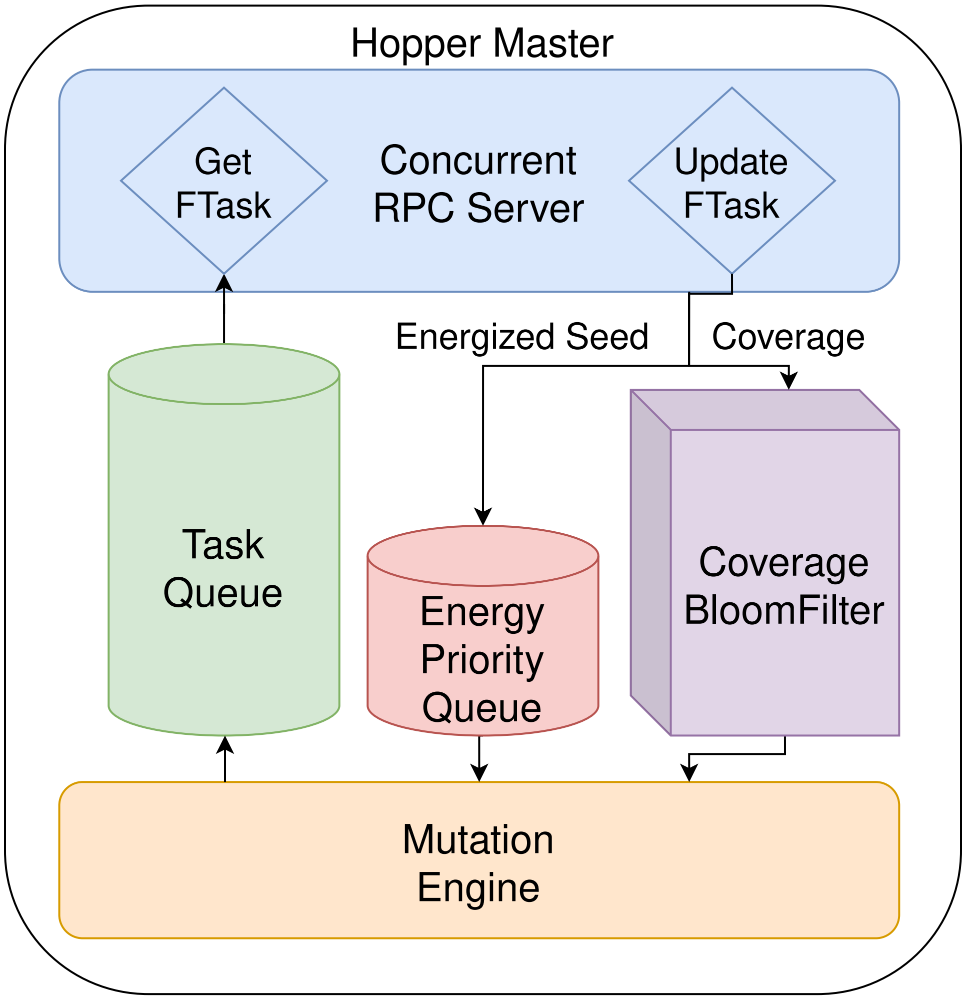


Figure 3.2: Hopper Master (relative scale of rough memory usage)

1. Receive Get FTask: Grab task from Task Queue, send task.
2. Receive Update FTask: Check that FTask exists, update state of task:
    (a) Update Coverage BloomFilter [24]
    (b) Add Energized Seed to Energy Priority Queue
3. Mutation engine grabs the most energized seed:
    (a) Mutate seed
    (b) Scale by a task queue available capacity factor
    (c) Put newly generated seeds into task queue

3.3.1 BloomFilter Deduplication

Hopper uses bloom filters as mentioned previously. According to Wikipedia:

"A Bloom filter is a space-efficient probabilistic data structure, conceived by Burton Howard Bloom in 1970, that is used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not – in other words, a query returns either 'possibly in set' or 'definitely not in set'." [24]

A bloom filter (BF) is a commonly used data structure in distributed caching systems [26] for it's space-efficient nature and probabilistic guarantees. It fits perfectly with the problems faced by a Distributed fuzzer, where we're attempting to synchronize potentially terabytes of data while still being space efficient. At the core, we don't care to retain every single seed that we've ever tested, however we do want to know whether we've seen this seed before. And therefore, a regular set would not make sense in this scenario because we'd run out of memory far before we hit the coverage plateau.

Precisely two bloom filters are kept in Hopper. One BF is used to de-duplicate similar coverage seeds, the data being serialized here are all the edge data received from a coverage run. And the other is a BF used for seed deduplication, where the seed itself is hashed into the BF. Both BFs use the same initial configurations, with an estimation formula show in Figure 3.3 for the number of bits m in the set, and number of hash functions k. Given that p is the probability of false positives and n is the number of estimated elements:

Figure 3.3: Bloom filter estimation

Given:

$$n = 100,000,000$$
$$p = .001$$

Then,

$$m = ceil(-1 * n * log_2(p)/log_2(2)^2)$$
$$k = ceil(log_2(2) * m/n)$$

3.3.2 Scheduling

Scheduling of tasks is a two-step process. First, a seed must be energized, it's then placed in the energy priority queue to prioritize more interesting inputs. Finally, when the task queue drops below half capacity, the mutation engine will grab the most energized seed from the priority queue, mutate it, and place them in the FIFO task queue. The reason we wait to mutate seeds until half capacity is because we want to allow enough time for the energy priority queue (EPQ) to fill up. This acts a pseudo-loadbalancer to keep the queues fair and provides enough buffer time to run the mutation function. If we didn't do this, when we grab a seed from the EPQ, mutate it into 100 new seeds, the Task Queue might already be full, and then when we scale by available capacity we will only add one seed. The energy assigning algorithm is a big factor to choosing interesting inputs, as shown in Figure 3.4:

Figure 3.4: Energize Seed

```
Require: $seed, crashed, allSeeds$
  $maxEdges \leftarrow max(EdgeCoverage(allSeeds))$        ▷ the max number of edges covered
  $edges \leftarrow EdgeCoverage(seed)$        ▷ number of edges covered by this seed
  $energy \leftarrow min(1, edges/maxEdges)$        ▷ Energy Range: (0, 1)
  if seed crashed then
    $energy \leftarrow energy \times 2$
  end if
```

3.3.3 Mutation Engine

The Mutation Engine in Hopper is quite standard, and a general algorithm is used to implement the mutation function, typically called the Mutator. The Mutator can be easily changed, because of Hopper's modular design, and it does not affect the rest of the system. The Mutation Engine is running completely asynchronously from the rest of the Master and is in charge of mutating seeds and controlling seed flow through the system.

A *Havoc* level can be chosen for the Mutation Engine at the beginning of the campaign when starting up Hopper. Havoc in the context of the Mutation Engine is just the number of times to run the mutation function (Mutator) on the input. A higher havoc level can yield wider exploration, but could also skip some interesting inputs by over-mutating the corpus in a few iterations. There's always a perfect balance between a significant mutation, while still being small enough to catch the most interesting inputs. Typically, havoc is increased for larger seed sizes. For example, when fuzzing *libpng*, the seeds might be megabytes large and doing a havoc level of 5 might be too little for that seed size. Sufficient experimentation should be done to determine the optimum havoc level for a particular target. But first, before a seed can be passed through the Mutator, there are a few filtering steps that occur. These filters are:

- If the seed has already been seen, it's deduplicated and not added to the task queue
- If an Update FTask is received and the coverage BF is not updated, then we already have a seed that produces the exact same coverage. Thus, we can completely get rid of that seed before it's added to the EPQ.
- If the seeds' coverage is so low that the energy fraction rounds down to zero, then the seed is not mutated once. Therefore, it's essentially thrown out by the energy algorithm.

Given that a seed has passed these filters, Hopper uses a default standard byte-level Mutator. Thus, Given a havoc level of 5, the following default Mutator will be applied to that seed five times, forming a new seed and thus a new FTask:

- **MUT**: change one random byte to a randomly generated byte
- **DEL**: delete a random byte
- **ADD**: add a random byte
- **SWP**: swap two random bytes
- **FLP**: flip all the bits in a random byte, ex: 10011011 $\rightarrow$ 01100100
- **REV**: reverse the bits in a random byte, ex: 10011011 $\rightarrow$ 11011001

3.3.4 Logging & Reporting

Hopper keeps track of all the following information throughout the fuzz campaign:

- Fuzzing instances (*its*): Total completed FTasks
- Crashes: Total crashes and their associated crash reports
- Seeds: Total number of seeds generated (this number is always greater than *its*)
- Nodes: Current active nodes in the cluster
- Coverage: Bloom filter that acts as a coverage map and the max number of edges covered by a single input
- Unique Paths: Number of unique paths explored in the PUT, told by the number of times we had to update the coverage BF
- Unique Crashes: Number of crashes that have a different coverage hash

Hopper can produce log reports at any point during the campaign. These reports contain all the information outlined above, as well as specific crash information gathered by each individual Node. Hopper can also be configured to produce a report on a set interval, this can be useful when gathering periodic data. This is how a lot of Hoppers long-term observability is provided.

Hopper also has a built-in Text User Interface (TUI), to display all the information shared above, except for crash specifics. This is done as an easy way to keep track of the cluster and the campaign from the Master. The TUI also has direct control of the Master, which means it can receive a command to gracefully stop the campaign and produce a report. This will send a kill signal to all the fuzzing nodes, denoted by a negative FTaskID. Then it will produce one final report of the campaign up to that point, and finally kill the Master.

3.4 Node

A Hopper Node's job is to run the PUT, gather and parse coverage, and report coverage/crash data to the Master. Each Node runs a main Fuzz loop. Nodes are fairly synchronous, with a few sections of parallelism for logging crashes and clean-up. But generally we keep each instantiation of a Node synchronous such that we can more easily reason about it as a discrete unit of computation. Nodes connect to the master through TCP/IP on a long-lived connection, this connection is kept open through their lifetime to avoid repeated handshakes. If the Node ever looses connection, it will indefinitely rety to connect. These TCP/IP connections are used to make the two RPC calls to Get FTask and Update FTask.

At the start of every campaign when starting up Hopper Nodes, each is given a unique NodeID, usually this is just a number that the Master uses to uniquely identify nodes. Each Node is also given a compiler orchestrated version of the PUT ELF (usually shared if nodes are on the same machine). This compiler-level orchestration is done to gather edge coverage through each run of the PUT. Specifically, Hopper uses LLVM SanitizerCoverage [16] to gather coverage, along with LLVM's built-in Address Sanitizer to detect crashes and build analysis reports that are then logged on the file-system upon a crash. Nodes extract edge coverage and generate a hash of the edge coverage graph that is then sent as part of the Update FTask used by the Master to de-duplicate seeds based on similar edge traversals.

3.4.1 Executor

Each Node is running one main fuzz loop, outlined by purple arrows in Figure 3.5. The two notable parts of the fuzz loop are the Executor and the Coverage Parser.

The Executor is concerned with setting up the environment for the PUT to run correctly. It will create environment variables and craft the exec command that runs. This is predefined in the beginning of the fuzz campaign and the Executor is in charge to make sure that the environment is configured correctly. Then it will fork and exec the process to begin the out-of-process fuzzing. The PUT will then finish executing with the provided input. And the rest of the fuzz loop is handed off to the Coverage Parser.

3.4.2 Coverage Parser

The Coverage Parser is concerned with reading the coverage and crash output produced by the PUT for a particular execution. It will read and parse the edges produced, hash them into four 64bit hashes that are then sent to the Master and will be used for the coverage bloom filter. The reason we compute these hashes on the Nodes and not the Master is because a 4 byte hash is usually smaller than the entire edge graph data, therefore saving on RPC call size and reducing per Node bandwidth. As part of its parsing, it also analyses the output of the PUT and verifies if the sanitizer produced a crash report due to memory corruption, if so,

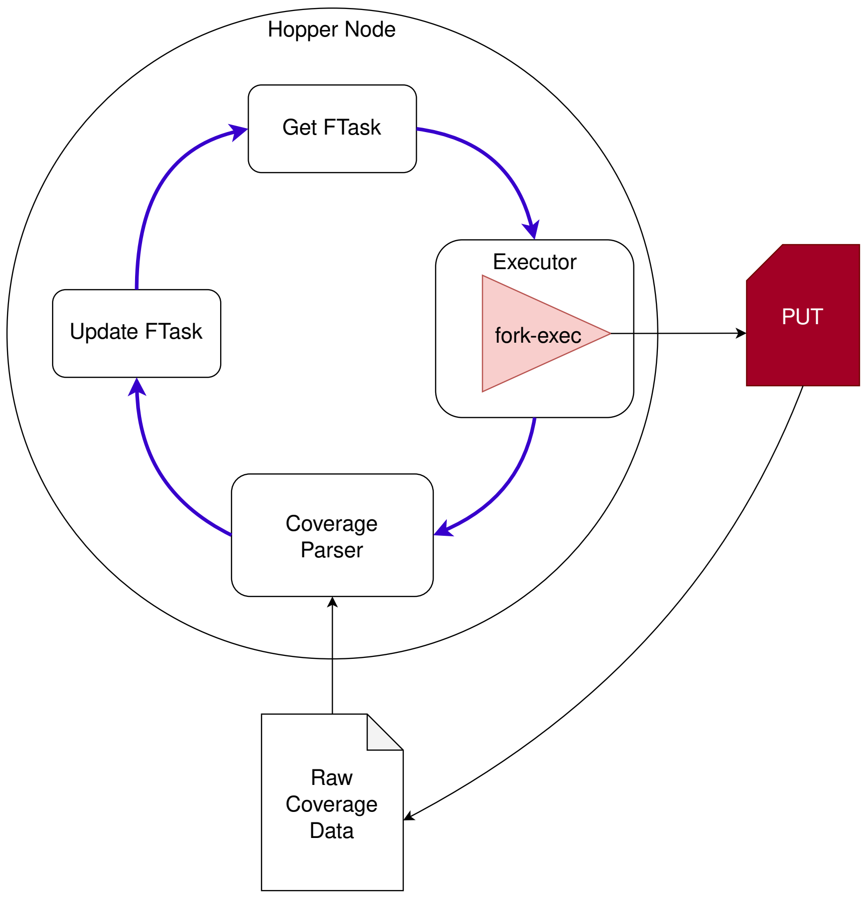


Figure 3.5: Hopper Node (relative scale of rough memory usage)

that is also reported back to the Master in the Update FTask RPC call.

Overall, the data in the Update FTask RPC call is as follows:

- NodeID: used to identify the Node (used by the Master to track unique Nodes connected)
- Ok: Execution completed correctly (If not Master will retry jobs that are not Ok)
- CovHash: the 4 byte coverage hash, this serves two purposes. One, the Master doesn't have to calculate the hash. Two, the data send over the network is reduced because 4 bytes is substantially less than the raw coverage graph data.
- CovEdges: The number of Edges the input covered.
- Crash: Did a crash happen
- CrashType: The type of crash, as determined by the Sanitizer (empty if no crash observed).

3.4.3 Logging

Nodes keep track of how many crashes and the types of crashes they've observed, this is separate from the clusters' data. It uses this data to produce unique crash information related to its NodeID which is assigned to it upon spawning the Node. When a node is instructed by the Master to log a crash because the Master has deemed the crash unique. It will log two files, one containing the seed that produced the crash and the other is the sanitizer report produced. An address sanitizer report will typically contain information on the stack frame, location in memory where the crash happened, and what caused the corruption.

At the end of the campaign, each Node will have a directory that contains the pairs of outputs for each crash. Each pair of outputs has matching CrashIDs used to generate the file names. This makes it easy to find the exact crash report and crash input for the bug that was reported by the Master. Along with the stack frame and the ASAN output, it should then be trivial to triage and find the bug in the PUT.

4 Implementation

4.1 Overview

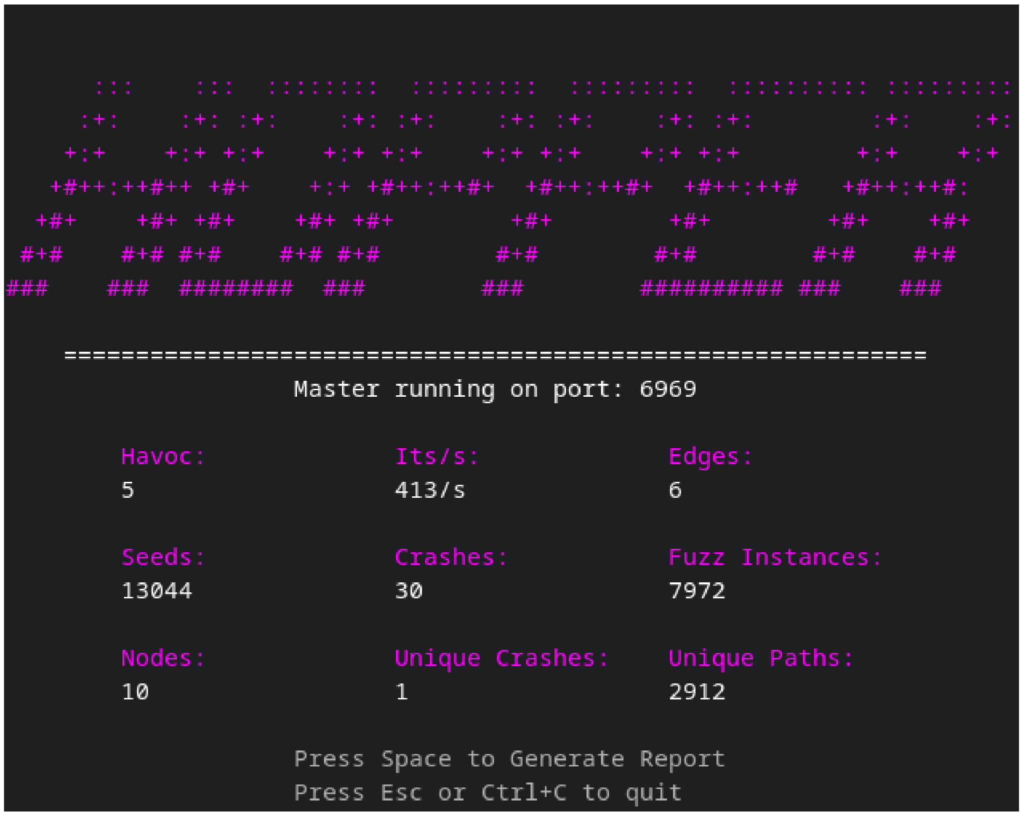
Hopper was implemented as a proof of concept in Golang version 1.19, mostly for its great compatibility and ease of use in the context of distributed systems. Scaling a system written in Golang is nearly trivial and sometimes completely uninvolved, as the go-routine scheduler will naturally take care of most concurrent needs. While Go has some definite downsides with its limited generics and abysmally unpleasant lack of functional operators and paradigms [19]. It fit the problem that Hopper was trying to solve perfectly, making it the best tool for the job. It's important to recognize that go threads aren't real threads, as in, they're not kernel threads. They are abstractions for work jobs that get assigned dynamically at runtime to a shared kernel thread pool. This is why they're usually called go routines instead of threads. But I will refer to them as threads from now on, as it's simpler to think of go-routines as threads.

The implementation follows the design very closely, and it can even be difficult sometimes to draw the lines between implementation and design detail. Just like in the design there are two main processes running on the Master, these are implemented as two separate threads. Naturally, the main thread of Hopper is handling RPC requests, where each RPC call is concurrently handled by spawning a new thread to handle each new request. Another thread is spawned called the *mutGenerator*, essentially functions as the Mutation Engine as mentioned previously.

There's a slew of command-line arguments and environment variables that Hopper recognizes. These are used to handle logging intervals, set havoc levels, or even specify the port at which the Master is supposed to run. The most important one being the *HOPPER_OUT* environment variable that specifies the location to save the logs to. This is by default set to the current directory. However, the complete list of these such commands and environment variables is on Hopper's GitHub, along with explanations and usage examples.

4.2 TUI

Given that the Master is the command center for the whole campaign, it felt necessary to build a nice kind of UI to look at for observability and to keep track of where we are on the coverage exploration curve.



```
      :::    :::  ::::::::  :::::::::  :::::::::  :::::::::: :::::::::
     :+:    :+: :+:    :+: :+:    :+: :+:    :+: :+:        :+:    :+:
    +:+    +:+ +:+    +:+ +:+    +:+ +:+    +:+ +:+        +:+    +:+
   +#++:++#++ +#+    +:+ +#++:++#+  +#++:++#+  +#++:++#   +#++:++#:
  +#+    +#+ +#+    +#+ +#+        +#+        +#+        +#+    +#+
 #+#    #+# #+#    #+# #+#        #+#        #+#        #+#    #+#
###    ###  ########  ###        ###        ########## ###    ###

    ============================================================
                    Master running on port: 6969

        Havoc:              Its/s:              Edges:
        5                   413/s               6

        Seeds:              Crashes:            Fuzz Instances:
        13044               30                  7972

        Nodes:              Unique Crashes:     Unique Paths:
        10                  1                   2912

                    Press Space to Generate Report
                    Press Esc or Ctrl+C to quit
```

Figure 4.1: Hopper TUI

To provide universal compatibility, the decision was made to make it a Text User Interface. The TUI was implemented using a library called Bubble Tea [12]. It provides nearly all the information that the Master has access to, as can be seen on Figure 4.1. The TUI updates on a one-second interval with the true stats. It's kind of cool watching the iterations per second and the Fuzz Instances go up live as we spin up a campaign.

4.3 Hashes

The implementation of Hopper uses two hash functions:

- MD5 Hash [29]: MD5 is used to compute the seed hash, which the becomes the FTaskID. The MD5 hash function was chosen because it's a standard algorithm, with relatively low collision rate, and fast enough for our purposes.
- Murmur Hash [30]: Murmur Hash is used to compute the four 64-bit hashes for the coverage. It was used because the original implementation of the Bloom filter implementation used it, however the reason is that it's a simple hash function, and it's fast to compute. Which is perfect for the bloom filter since it's doing roughly ten of these hashes per new input.

4.4 Bloom Filter Implementation

The bloom filter implementation that Hopper uses is a modified version of the open source project by *bits-and-blooms* [15] and the bloom filter is backed by the same *bits-and-blooms* project's Bitset [14]. Most of the functionality was kept the same, except part of the interface had to be dissected such that the BF hashes could be computed on the Fuzzing Node instead of on the Master to optimize network throughput and make coverage data ephemeral. This was a substantial speed improvement, as half the work of using a bloom filter is then distributed amongst the Nodes. The Master is then able to directly update the BF without having to compute the hashes.

As mentioned previously, our bloom filter uses a variation of murmur hash, this hash function is built directly into hopper and shared among the Master and fuzzing Nodes. The Master will rarely compute the hashes required by the coverage bloom filter. This gives us another dimension of scaling because each Node cuts down the cost of doing relatively expensive operations on the Master. By the time the Master gets the coverage data, it's already been pre-processed, parsed, and hashed by the Nodes. Making it feasible to continuously synchronize coverage with the Master.

4.5 Channels & Queues

There are two queues in Hopper, the Task Queue and the Energy Priority Queue. Hopper leverages Golang primitives like go channels to implement the FIFO queue. This is how the Task Queue was created, channels provide a natural way of abstracting a queue while still providing one reader and one writer guarantees. Making it easy to do concurrent operations that required access to the Task Queue. Channels are at the underpinning of the scheduling and managing of Hoppers queues. Specifically, a buffered channel of size 10,000 was used as the Task Queue.

The EPQ is implemented using the standard heap implementation from the Go std container library. With a bit of boilerplate code to turn the heap into a max heap, the EPQ behaves exactly as what we'd expect a priority queue to behave. This is then used with floating point numbers to represent the energy and therefore rank in the priority queue. The EPQ size is set to 5,000 as this is exactly half of the size of the Task Queue as shown in Figure 3.2. After much experimentation, this seemed to provide the best results for clusters of 200+ nodes, with no capacity boundaries being reached.

Each Hopper Node is also running an asynchronous buffered task queue, implemented as a buffered go channel. A thread is spawned to keep this task buffer queue full, this is done to maximally utilize each Nodes time fuzzing and prevent the main fuzzing loop from being throttled by RPC calls to the Master to get a task. This is done purely for optimization reasons, however it can be considered part of the core integration on the Node. The queue is only buffered for 5 items, as this is typically more than enough time for the Node to grab a fuzz task and run the input. By the time it's done, there will definitely be another fuzz task in the queue.

4.6 Demos & Scripts

Many fuzzers can be difficult to configure and run correctly. Usability is an often overlooked issue with new software proposals in fuzzing. Hopper prides itself on being easy to use, while still providing the core utilities of a fuzzer and the parallelism of a distributed system. Hopper provides a set of tools to containerize, distribute, and arbitrarily spawn fuzzing agents. Apart from the modular design of the core Master and Node Go software, we also provide scripts to facilitate campaigns. We consider this part of the contribution that Hopper is making to fuzzing, and shifting towards infrastructure-aware software proof of concepts. Just because it's a proof of concept, doesn't mean it should be **impossible** to use. An easy to use fully dockerized demo is available on the Hopper GitHub.

Substantial effort was placed into making containerization and distribution scripts for Hopper. These scripts were crafted mainly to run the evaluation, and so there's a lot of knowledge that can be gleaned off of reading the scripts and docker files for how to run a campaign with Hopper. This also allows for reproducibility of results, due to its standardized running environment and native build scripts. Hopper has a core docker image that contains both the Master and Node build, this makes it easy for anyone to take it and make few modifications to get it to fit their use-case. Dockers multi-stage builds were used to compose multiple images into a docker environment perfectly suitable for fuzzing specific targets.

A python script was also created to pull and compile any version of the GNU standard binutils. This script injects all the necessary flags and compiler tooling required to run a Hopper campaign. These were heavily leveraged when deciding what binutil to primarily focus on. Bash scripts were also provided to distribute, setup, and spawn containers for a Master or Node. These scripts were directly used in the evaluation of Hopper, again, available on GitHub.

5 Evaluation

5.1 Overview

All experiments with Hopper are from the main branch on GitHub and were done on Cloudlab [6]. Graciously provided by our very own FLUX Research group. Specifically, all evaluations were conducted on **c6525-100g** machines. Both the Master and the fuzzing Nodes ran on these machines on a 10Gbit/s (full duplex) subnet. All communication was done through the internally provided Cloudlab experiment network. All nodes communicated over RPC, even the Nodes running on the same system as the Master. This is done for uniformity and to consider the possibility of running nodes on a one to one, machine to node configuration. In Table 5.1 the specs are outlined:

c6525-100g Machine Specs	
Architecture	x86 64bit
OS	Ubuntu18-64-X86
Processor	AMD EPYC 7402P @ 2800MHz
Cores	24
CPU Threads	2
Memory Size	131072 MB (131 GB)

Table 5.1: Evaluation Machines

5.2 Linear Scaling

Five experiments were run to evaluate the efficacy of Hopper on its ability to scale linearly with the number of nodes in the cluster. Five c6525-100g machines were used to run all the experiments, to retain consistency among all experiments. Each experiment, I increase the number of active nodes in the cluster in each subsequent experiment, starting at 45 going up to 240 distributed evenly among all 5 c6525-100g machines. Each node is labeled as node{1 - 5}, the Master is always running on node1. The target was binutils-readelf version 2.40 [9], this is the newest version having been released January 16th 2023. The Master was configured with a havoc level of 20, and nodes saved crash information on a shared experiment file system. For the final 240 node cluster, we only put 45 on the Master because by that point we're approaching the limits of the hardware and if we put more fuzzing nodes on that machine it'll eat into the performance of the Master. The distribution of nodes can be seen in Table 5.2.

Node Count	*node1*	*node2*	*node3*	*node4*	*node5*
45	9	9	9	9	9
95	19	19	19	19	19
145	29	29	29	29	29
195	39	39	39	39	39
240	45	49	49	49	49

Table 5.2: Node Distribution

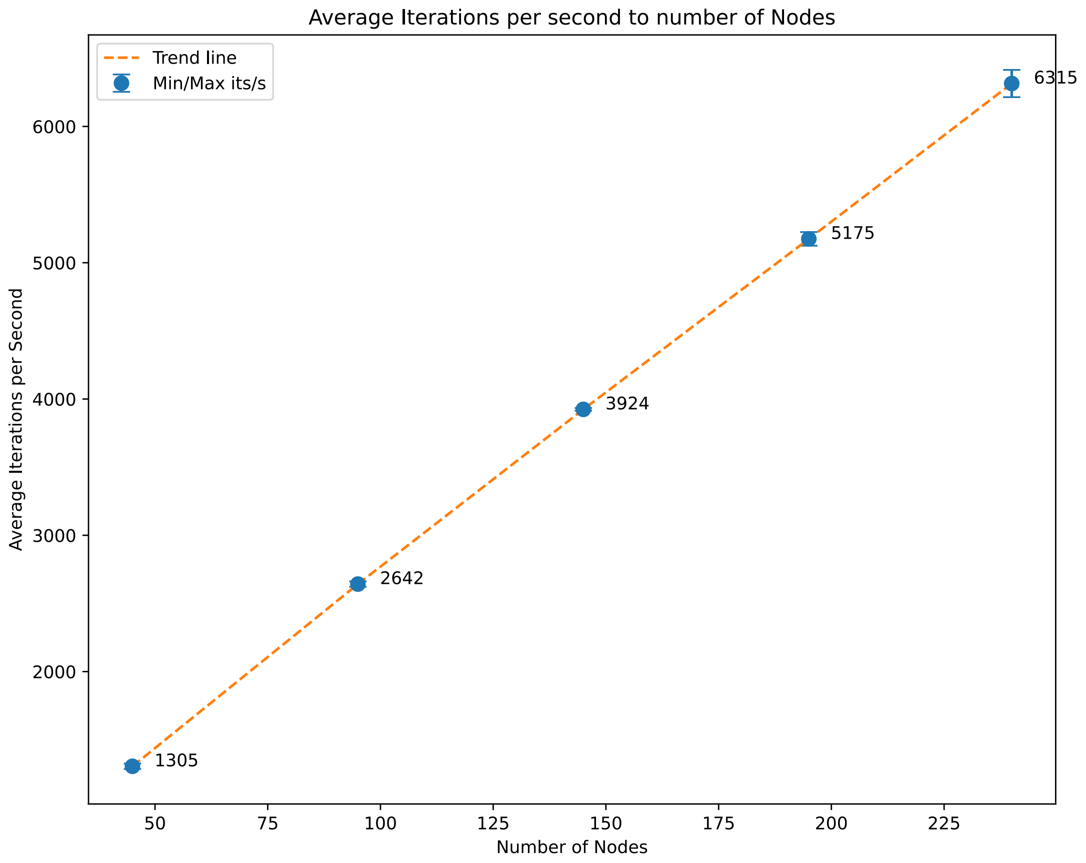


Figure 5.1: Ave. Its/s vs Node Count

As we can see from Figure 5.1 it's clear that as we scale the number of nodes, we notice a linear increase in the average iterations per second. This concurs with our hypothesis directly, we observe that if a proper distributed system is implemented into a fuzzer. One can expect a linear increase in iterations per second, defined as the number of fuzz jobs completed at a certain time interval. Even though we only evaluate readelf here, as mentioned previously, a build script was made to download and build any binutil version. Thus, Hopper can be tested against any of the standard binutils. Readelf was chosen for its simplicty and ease to fuzz.

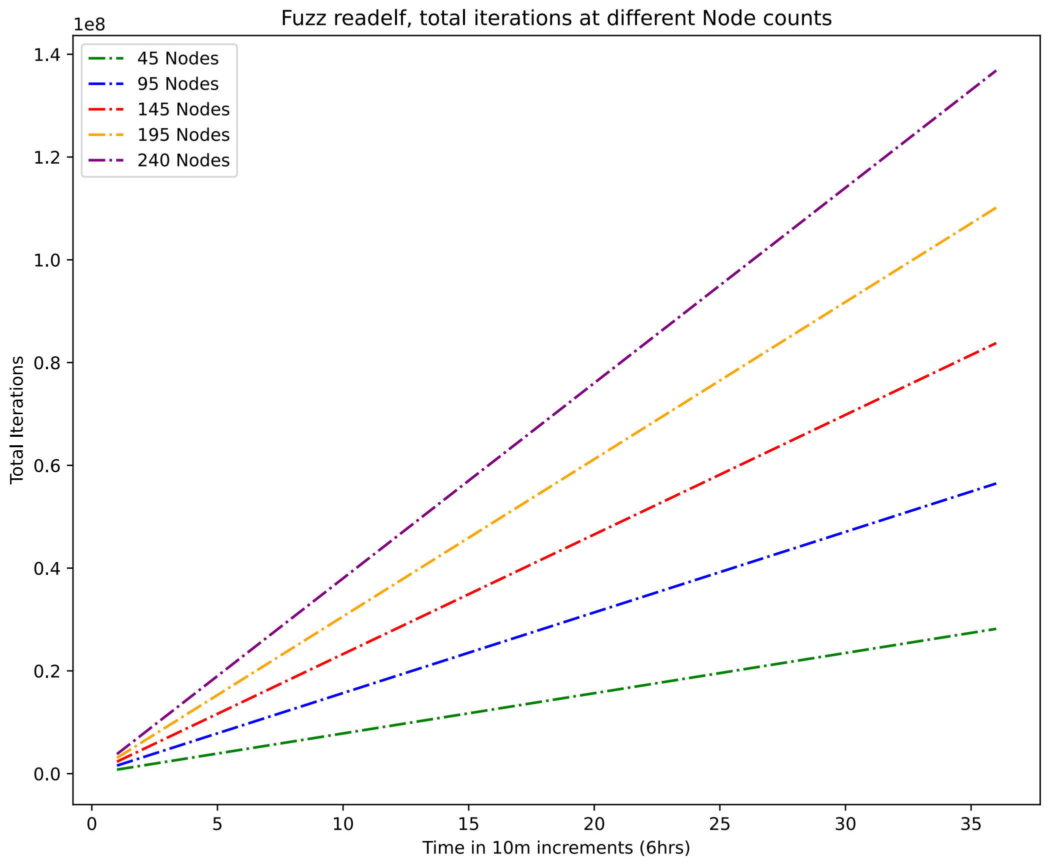


Figure 5.2: Total Iterations over time per Node count

The data gathered appears exceptionally nominal, as can be seen in Figure 5.2, but it's what we would expect from a consistent and stable system. We observe a clear linear distinction between each Node count and the number of iterations. However, for longer than 6 hour campaigns, we do observe a noticeable drop in iterations per second. This has remained undeciphered, on average at the 15 to 20 hour mark, we start to notice a 200 to 300 drop in iterations per second. This appears to be correlated with the size of the system, as it only appears to happen when running at near-machine-limit node counts, such as 240. The best hypothesis for this is that the Mutator is increasing the seed size and favoring larger seeds for their coverage, and therefore increasing overall traffic in the system.

5.3 Coverage Plateau

We've also observed the ever so infamous coverage plateau on several runs. Notably, our 240 node 16hr and 195 node 16hr runs under the same configurations described previously, as can be seen in Figures 5.3 and 5.4

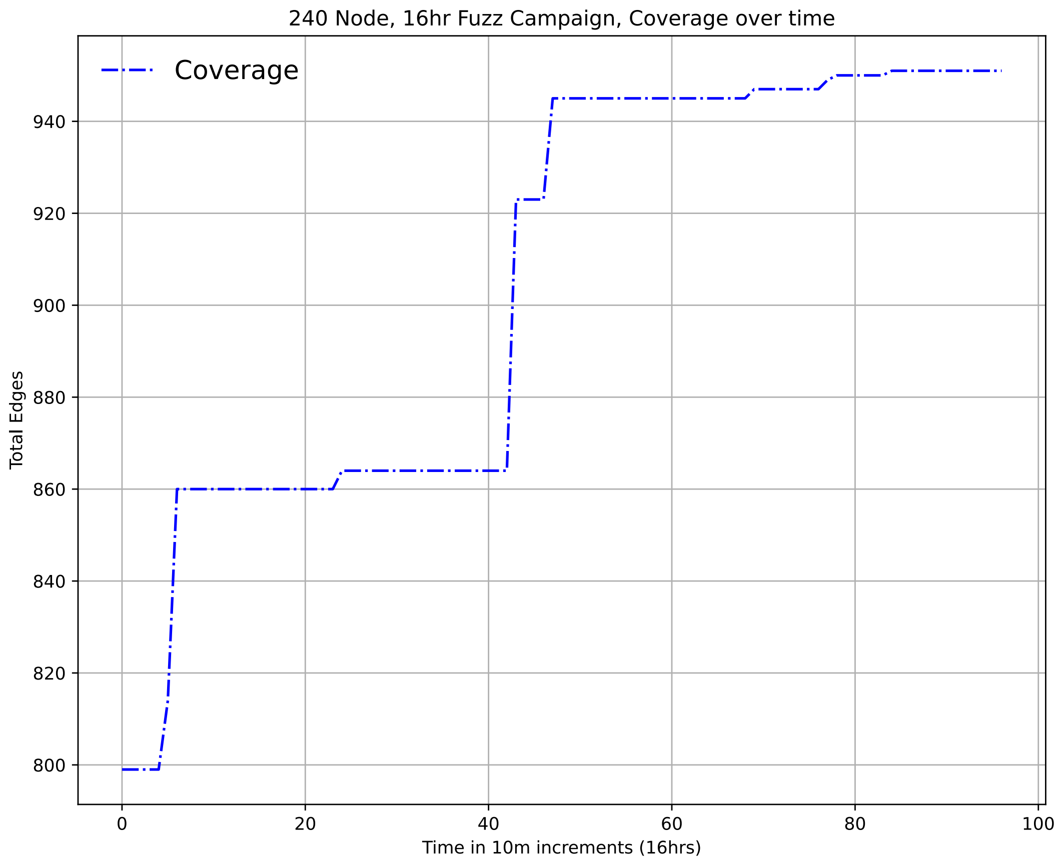


Figure 5.3: Coverage over time 240 Nodes

As we can see from Figure 5.4 a shorter run was able to get slightly more edge coverage. This is due to a really lucky mutation, underscoring the way that our randomly seeded mutation algorithm works. However, the largest number of edges we were able to reach was actually 1075, out of the roughly 2400 edges in readelf according to SanitizerCoverage utility. This was acquired on a really lucky 240-node campaign, after only 5 hours of fuzzing, as we can see in Figure 5.5.

In Figure 5.5 we also observe the linear scaling difference of four different fuzz campaigns over the span of 5 hours. The graph is precisely what we would expect from a linearly scaling fuzzer. Individual coverage

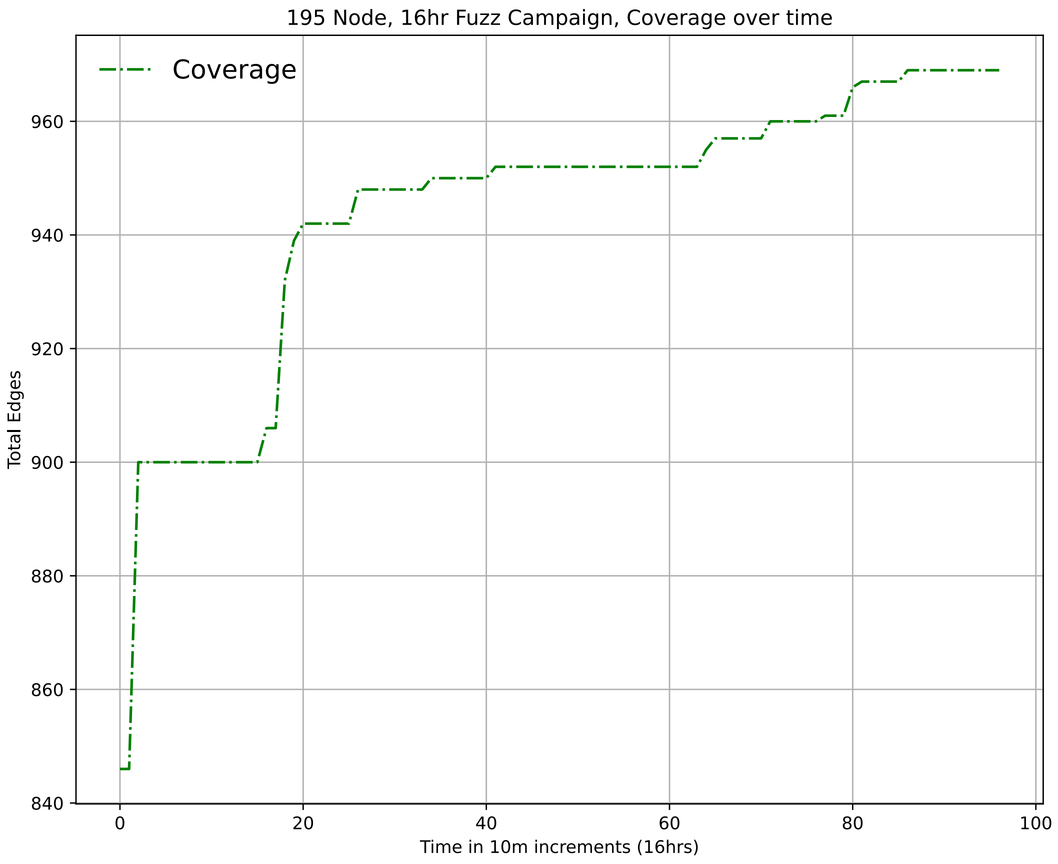


Figure 5.4: Coverage over time 195 Nodes

is logarithmic, however we observe that the plateaus are higher for campaigns with more fuzzing Nodes. In the 95 Node experiment, we see that it reached a higher edge count than the 145 node run initially, but eventually it was surpassed.

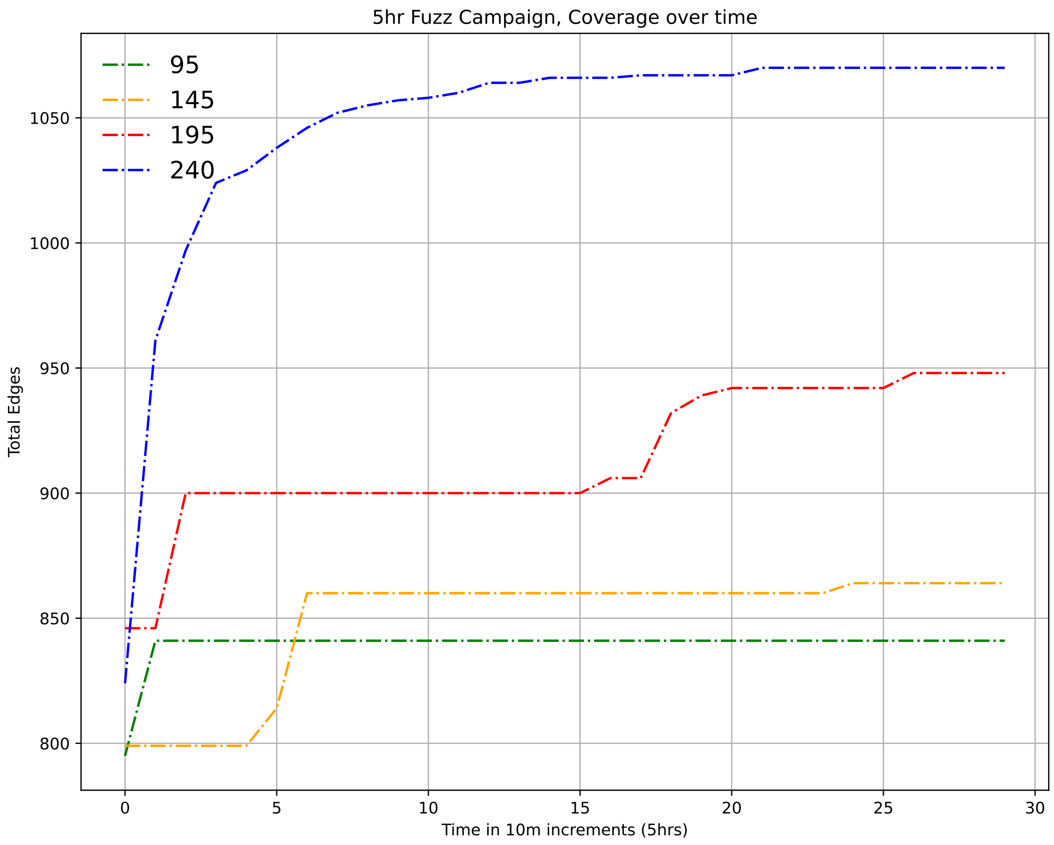


Figure 5.5: Coverage comparison to number of Nodes

5.4 Crashes

After running 27 different fuzzing campaigns, all ranging in different number of Nodes, we did find 13 uniquely different crashes. However, these crashes correspond to bad inputs and not necessarily a bug in the code. Most of them were an early exit that returned a non-zero exit code. None of these triggered an ASAN report, so it's not conclusive whether we found a bug. We believe that this is largely a function of our mutation, energy strategy, and our choice in the newest version of readelf. If we had fuzzed an older version of readelf, we could have potentially discovered some already patched CVEs. However, the focus of Hopper's design and implementation was to focus on its distributed infrastructure and scalable design. We will leave further investigation in this area to another publication.

Conclusion

From the design, implementation, to the evaluation, we were able to see that our design for a distributed fuzzer has proven to scale linearly with the number of Nodes. The Master-Node configuration does a good job at facilitating communication and synchronization between nodes. Hoppers core design principles with bloom filter deduplication and seed deduplication have allowed more significant iterations. Such that, any improvements in iterations per second made in the future might yield more significant returns. Albeit, the effect and importance of these deduplications on fuzzing campaigns still needs to be proven through further investigation. Hopper has proven its efficacy in scaling, and its core modular design will allow the greatest parts of its design to be used independently in future works. Hopper was built with an awareness of current state of the art fuzzers, making it a great starting-point for further research into the space. It can serve as an early blueprint for building scalable distributed fuzzers.

The implementation on GitHub as of April 24th 2023, has 67 stars and growing. It seems to have caught the eye of some Open Source maintainers and fuzzing enthusiasts. Hopper is still in its early stages and much more will be done in the future to bring it to being a competitive fuzzer in all aspects, not only its distributed design. I intend to continue developing Hopper, by improving its distributed design, and core fuzzing features.

Future Work

There's lots of work that can be done atop what's been built with Hopper. One notable thing that can be improved in the current design of Hopper's Node Update RPC protocol, is to do batching. The nodes cause a lot of network congestion, it might be beneficial to introduce a batching update system for the Nodes to batch update FTasks, instead of one update per RPC call. This is a feature currently being worked on, but not fully implemented into Hopper.

Another possible path is to explore redesigning the network overlay of how Hopper communicates with Nodes, instead of a Master-Node configuration. The designs could be merged into a full-mesh system with a single distributable Node. Although this would require a significant change in the implementation, a substantial portion of the design would remain the same, except that it would be merged into one Node type. Where each node synchronizes its coverage with other nodes, Hopper's current implementation could be changed to achieve this goal with only a week worth of effort. A full-mesh network overlay will provide better fault tolerance and would remove the single point of failure that is the Master, due to this design having no Master. In this new design, Hopper Nodes would have all the state a Master has, and would have full observability over the cluster. However, a possible downside to this is that the amount of traffic would substantially increase, due to each node having to communicate with N-1 other nodes in the cluster to get coverage information. This would also make it significantly more difficult to do coverage and seed deduplication, because all the data is sparsely spread across the whole cluster and no node has a fully updated view of the cluster at any given moment.

Finally, there are many improvements that can be made to the core fuzzing functionalities that don't affect the distributed design. Such as, making Hopper an in-process fuzzer, improving the mutation algorithm, improving the energy calculation algorithm, or even just using a better structured RPC protocol like gRPC instead of the Golang std net/rpc. If these core fuzzing functionalities were changed for the better, this would only compound the design of Hoppers distributed infrastructure. Another possibility is to rewrite the fuzzer in Rust. Although we haven't hit the limit with number of concurrent nodes, at a certain point Golang's GC will cause problems with the handling of so many concurrent fuzzing Node connections that it might slow down the Master. This could be catastrophic for the iterations per second, as throttling the Master means throttling the system as a whole.

References

[1] F5 Cyentia Institute. *The Evolving CVE Landscape*. Last accessed 05 Apr 2023. 2023. URL: https://www.f5.com/labs/articles/threat-intelligence/the-evolving-cve-landscape.

[2] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *OSDI’04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150.

[3] Fadi P. Deek and James A. M. McHugh. “The GNU project”. In: *Open Source: Technology and Policy*. Cambridge University Press, 2007, pp. 297–308. DOI: 10.1017/CBO9780511619526.009.

[4] Pouria Derakhshanfar and Xavier Devroey. “Basic Block Coverage for Unit Test Generation at the SBST 2022 Tool Competition”. In: *2022 IEEE/ACM 15th International Workshop on Search-Based Software Testing (SBST)*. 2022, pp. 37–38. DOI: 10.1145/3526072.3527528.

[5] Zhen Yu Ding and Claire Le Goues. “An Empirical Study of OSS-Fuzz Bugs”. In: arXiv, 2021. DOI: 10.48550/ARXIV.2103.11518. URL: https://arxiv.org/abs/2103.11518.

[6] Dmitry Duplyakin et al. “The Design and Operation of CloudLab”. In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. July 2019, pp. 1–14. URL: https://www.flux.utah.edu/paper/duplyakin-atc19.

[7] Andrea Fioraldi et al. *AFL++*. 2022. URL: https://github.com/AFLplusplus/AFLplusplus.

[8] Andrea Fioraldi et al. *AFL++ Docs*. Last accessed 14 Apr 2023. 2023. URL: https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/afl-fuzz_approach.md.

[9] GNU Project - Free Software Foundation. *Index of /gnu/binutils*. Last accessed 24 Apr 2023. 2023. URL: https://ftp.gnu.org/gnu/binutils/.

[10] Robert Griesemer, Rob Pike, and Ken Thompson. *Golang, go.dev*. 2023. URL: https://go.dev/.

[11] Graydon Hoare. *Rust, rust-lang.org*. 2023. URL: https://www.rust-lang.org/.

[12] TJ Holowaychuk and Christian Rocha. *Bubble Tea, GitHub*. Last accessed 16 Apr 2023. 2023. URL: https://github.com/charmbracelet/bubbletea.

[13] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation". In: *CGO*. San Jose, CA, USA, Mar. 2004, pp. 75–88.

[14] Daniel Lemire. *bitset, Github*. Last accessed 16 Apr 2023. 2023. URL: `https://github.com/bits-and-blooms/bitset`.

[15] Daniel Lemire. *bloom, Github*. Last accessed 16 Apr 2023. 2023. URL: `https://github.com/bits-and-blooms/bloom`.

[16] Chris Lattner LLVM and Vikram Adve. *SanitizerCoverage — Clang 17.0.0git documentation*. Last accessed 16 Apr 2023. 2023. URL: `https://clang.llvm.org/docs/SanitizerCoverage.html`.

[17] Barton P. Miller, Lars Fredriksen, and Bryan So. "An Empirical Study of the Reliability of UNIX Utilities". In: *Commun. ACM* 33.12 (1990), pp. 32–44. ISSN: 0001-0782. DOI: `10.1145/96267.96279`. URL: `https://doi.org/10.1145/96267.96279`.

[18] Ruixiang Qian et al. "Investigating Coverage Guided Fuzzing with Mutation Testing". In: *arXiv preprint arXiv:2203.06910* (2022).

[19] Luciano Remes. *Why Golang is Almost Perfect*. Last accessed 16 Apr 2023. 2023. URL: `https://www.lremes.com/posts/golang/`.

[20] Kosta Serebryany. "Continuous Fuzzing with libFuzzer and AddressSanitizer". In: *2016 IEEE Cybersecurity Development (SecDev)*. 2016, pp. 157–157. DOI: `10.1109/SecDev.2016.043`.

[21] Kosta Serebryany. *LibFuzzer authors leave the project*. Last accessed 14 Apr 2023. 2023. URL: `https://llvm.org/docs/LibFuzzer.html`.

[22] Shevchenko and Serebryany. *Google Centipede*. Last accessed 17 Jan 2023. 2023. URL: `https://github.com/google/centipede`.

[23] Bjarne Stroustrup. *C++*. 2023. URL: `https://en.wikipedia.org/wiki/C++`.

[24] Wikipedia. *Bloom filter*. Last accessed 15 Apr 2023. 2023. URL: `https://en.wikipedia.org/wiki/Bloom_filter`.

[25] Wikipedia. *Concurrency*. Last accessed 15 Apr 2023. 2023. URL: `https://en.wikipedia.org/wiki/Concurrency_(computer_science)`.

[26] Wikipedia. *Distributed Cache*. Last accessed 15 Apr 2023. 2023. URL: `https://en.wikipedia.org/wiki/Distributed_cache`.

[27] Wikipedia. *Distributed Computing Wikipedia*. Last accessed 14 Apr 2023. 2023. URL: `https://en.wikipedia.org/wiki/Distributed_computing`.

[28] Wikipedia. *Fuzzing*. Last accessed 17 Jan 2023. 2023. URL: `https://en.wikipedia.org/wiki/Fuzzing`.

[29] Wikipedia. *MD5*. Last accessed 16 Apr 2023. 2023. URL: `https://en.wikipedia.org/wiki/MD5`.

[30] Wikipedia. *MurmurHash*. Last accessed 16 Apr 2023. 2023. URL: `https://en.wikipedia.org/wiki/MurmurHash`.

[31] Wikipedia. *Open-source software Wikipedia*. Last accessed 17 Jan 2023. 2023. URL: `https://en.wikipedia.org/wiki/Open-source_software`.

[32] Xu Zhou et al. "UltraFuzz: Towards Resource-saving in Distributed Fuzzing". In: *IEEE Transactions on Software Engineering* (2022), pp. 1–22. ISSN: 1939-3520. DOI: `10.1109/TSE.2022.3219520`.

[33] Xiaogang Zhu et al. "Fuzzing: A Survey for Roadmap". In: *ACM Comput. Surv.* 54.11s (Sept. 2022). ISSN: 0360-0300. DOI: `10.1145/3512345`. URL: `https://doi.org/10.1145/3512345`.