applied sciences

Article

P-Fuzz: A Parallel Grey-Box Fuzzing Framework

Congxi Song , Xu Zhou *, Qidi Yin, Xinglu He, Hangwei Zhang and Kai Lu

College of Computer, National University of Defense Technology, Changsha 410073, China; congxi1994@sohu.com (C.S.); feidiyin@gmail.com (Q.Y.); hexinglu1992@gmail.com (X.H.); zhw1997118@126.com (H.Z.); kailu@nudt.edu.cn (K.L.)
* Correspondence: zhouxu@nudt.edu.cn

Received: 5 October 2019; Accepted: 20 November 2019; Published: 25 November 2019

Abstract: Fuzzing is an effective technology in software testing and security vulnerability detection. Unfortunately, fuzzing is an extremely compute-intensive job, which may cause thousands of computing hours to find a bug. Current novel works generally improve fuzzing efficiency by developing delicate algorithms. In this paper, we propose another direction of improvement in this field, i.e., leveraging parallel computing to improve fuzzing efficiency. In this way, we develop P-fuzz, a parallel fuzzing framework that can utilize massive, distributed computing resources to fuzz. P-fuzz uses a database to share the fuzzing status such as seeds, the coverage information, etc. All fuzzing nodes get tasks from the database and update their fuzzing status to the database. Also, P-fuzz handles some data races and exceptions in parallel fuzzing. We compare P-fuzz with AFL and a parallel fuzzing framework Roving in our experiment. The result shows that P-fuzz can easily speed up AFL about 2.59× and Roving about 1.66× on average by using 4 nodes.

Keywords: software testing; parallel fuzzing; AFL; vulnerability

1. Introduction

Fuzzing is an efficient method in software testing by providing unexpected inputs and by monitoring for exceptions [1]. In this way, thousands of security vulnerabilities are discovered by fuzzing [2,3]. According to the knowledge and information acquired from the target programs, fuzzing can be divided into white-box, black-box, and grey-box fuzzing. A state-of-the-art grey-box fuzzer American Fuzzy Lop (AFL) [4] collects the coverage information(edges in the target program are covered or uncovered) and stores it in a data structure bitmap to feedback further fuzzing.

Nevertheless, though fuzzers like AFL are simple and effective, they are still compute-intensive and cost a lot of CPU hours to fully test a program. Current novel works generally improve fuzzing efficiency by developing delicate algorithms [4–7]. Unlike all those works, we consider this efficiency problem of grey-box fuzzing in a different point of view. Instead of being limited to improving algorithms in a single computing node, we try to leverage more computing resources to parallelize the fuzzing tasks. In this way, we can trade resources for time to accelerate software testing. Our method is based on two observations. First, parallel computing is ubiquitous nowadays [8,9]. We can get massive, cheap computing resource easily (e.g., by using the Amazon spot instance [10]). Second, time is valuable in software testing and security. Considering the situation, a newly developed software is about to be released; it is worthy to spend more money than usual to make the release on schedule. Moreover, parallel fuzzing optimization is orthogonal with algorithm improving. Any improved fuzzing algorithm can be easily applied in a parallel fuzzing framework.

However, current parallel fuzzing approaches have drawbacks. Grid fuzzer [11] leverages grid computing to parallelize fuzzing by distributing fuzzing tasks statically. This method is not suitable for grey-box fuzzing, as work cannot be statically determined beforehand due to the feedback mechanism in grey-box fuzzing. Liang et al. [12] presented a distributed fuzzing framework which can manage computing resources in a cluster and can schedule resources to many submitted fuzzing jobs. However, it does not intend to accelerate a single fuzzing job. Roving [13] and the work of Martijn [14] can parallel AFL to fuzz a single program with distributed computing nodes. However, they only parallelize the nondeterministic mutation part of AFL and fail to parallelize the deterministic mutation part. Also, they do not synchronize the coverage information, which is a crucial part of grey-box fuzzers. PAFL [15] and Enfuzz [16] can parallelize several typical fuzzers together, which are proposed in 2018 and 2019. These two works are characterized by parallelizing diverse fuzzing tools to solve problems with the characteristics of different tools. However, these framework cannot share and sychronize tasks and resources across machines.

In this paper, we intend to design a parallel grey-box fuzzing framework and to leverage parallel computing to speed up the fuzzing process. We need to solve the following questions in this research:

1. How to synchronize and share fuzzing status, e.g., seeds (those test cases which trigger edges), the coverage information, etc. in a distributed system?
2. How to balance the workload to different computing nodes in the distributed system?
3. How to handle the data races and exceptions in a distributed system during fuzzing?

We implement a parallel grey-box fuzzing framework P-fuzz to solve these questions. P-fuzz consists of computing nodes to accelerate fuzzing. To share seeds and the coverage information which is stored in the bitmap, we leverage a key-value database. A computing node fetches a seed from the database and begins its fuzzing process: Firstly, the node mutates the seed to generate test cases. Then, the node sends test cases to the target program and monitors the execution of the target. When the node hits uncovered edges, it adds the corresponding test case to the database as a new seed and feedbacks this updated coverage information to the database to share benefits with other computing nodes. Also, we apply strategies to dynamically distribute fuzzing tasks to different nodes to achieve balance in the workload. To handle data races and exceptions, we analyze a set of specific cases and propose solutions for each case. In addition, we use the Docker [17] technique to build the environment of fuzzing framework and copy this environment to all computing nodes in the distributed system automatically. P-fuzz is capable of parallelizing fuzzing tools across machines, which is different from other frameworks just sharing fuzzing status in a file system. Also, it provides the scalability of parallelizing various fuzzing tools.

We evaluate P-fuzz in nine target programs and LAVA-M data benchmarks. The result shows that P-fuzz outperforms the AFL and Roving. Compared in bitmap density which reflects the coverage of target programs, P-fuzz enhances the bitmap density of AFL about $2.59\times$ and of Roving about $1.66\times$. It also triggers 49 crashes in target programs.

There are four contributions in this work:

- We design the method to share seeds and the coverage information with a distributed system to synchronize the fuzzing status.
- We design the method to balance workload by giving fuzzing tasks to different computing nodes dynamically.
- We handle data race cases and exceptions in the parallel fuzzing.
- We implement the parallel fuzzing framework P-fuzz to enhance the fuzzing efficiency.

2. Background

2.1. The Classification of Fuzzing

According to the knowledge and information acquired from the target programs, fuzzing can be divided into white-box, black-box, and grey-box fuzzing [18]. The white-box fuzzer has full knowledge of the source code (e.g., internal logic and data structures) and uses the control structure of the procedural design to derive test cases. In contrast, the black-box fuzzer does not have any knowledge of the target program; thus, it generates test cases randomly and swiftly. The grey-box fuzzer combines the efficiency and effectiveness of black-box fuzzers and white-box fuzzers, which masters limited knowledge of target programs. Currently, the grey-box fuzzing technique is practical and widely used in software testing and vulnerability detection as it is lightweight, fast, and easy to use [5].

The grey-box fuzzing process usually contains three steps:

1. An initial seed is selected from the prepared test cases set and mutated to generate a group of test cases.
2. Generated test cases are fed to target programs. At the same time, the fuzzer collects the coverage information (paths, edges, etc.).
3. The fuzzer utilizes the feedback information to select valuable test cases as new seeds (test cases that trigger new edges are considered as new seeds).

In this paper, we focus on improving the efficiency of grey-box fuzzing technique.

2.2. Details about AFL

As a grey-box fuzzer, AFL shows its benefits in effectiveness and efficiency. For sharing the fuzzing status, there are two things in AFL we need to care about in detail: the **seed** and **bitmap** data structure.

2.2.1. Seed

Seed indicates a test case which can trigger the fuzzer to traverse new edges. A queue in AFL is maintained to store the seeds. A high-quality corpus of candidate files will be selected as interesting seeds for further fuzzing.

2.2.2. Bitmap

Bitmap is a data structure which stores the coverage information of fuzzing. The bitmap size of AFL is 64 Kilobytes. A byte in the bitmap indicates an edge, which connects two or more basic blocks of the target program. The eight bits in a byte describe how many times this edge is covered. We use a tuple to express an edge; for example, if there are basic blocks A and B, then a *tuple (A, B)* means an edge from previous basic block A to the current basic block B. If a test case covers a new edge, the bitmap will record this changed coverage information by updating the corresponding byte. A mechanism to index the bitmap is shown as Equation (1). By simply reading the bitmap, AFL knows whether an edge is newly covered and decides to store or discard a test case [5].

$$(A \oplus B)\%BITMAP\_SIZE \tag{1}$$

Moreover, AFL runs deterministic mutations and nondeterministic mutations. Deterministic mutation strategies produce test cases and small differences between the non-crashing and crashing inputs [19]. Nondeterministic mutation strategies can make fuzzing achieve high coverage rapidly by random combining deterministic strategies. Roving [13] relies on the nondeterminism of AFL to cover more edges faster. However, for fuzzing a target program in which the input files are in a complex format, random mutations will destroy the format of files. Therefore, utilizing appropriate strategies to fuzz different programs is necessary.

2.3. The Discussion of Parallel Mechanism in Fuzzing

A computing node is capable of handling computing and of sending or receiving information with other nodes, which is the basic element in a distributed system. In fact, just putting a testing task on a multi-core machine or a distributed system but running it on a single computing node is underutilizing the hardware. At this time, parallel computing resources can make full use of hardware and can bring profit to low-efficiency fuzzing processes.

Two fuzzing frameworks extend the parallel mechanism in AFL. One is Roving [13], which is implemented by running multiple copies of AFL on multiple computing nodes, all of them fuzzing the same target. It benefits from the client–server structure which shares crashes, hangs, and queues of each client. Each computing node plays a role in a client or server. Every 300 s, clients update the fuzzing environment by uploading and downloading changes. The whole framework is scheduled by the central server. The other is the work of Martijn [14]. The main idea of this work is approximate to Roving, and the difference between them is the implementation.

Although the two frameworks utilize computing resources and parallelize the fuzzing progress, which makes each client benefit from each other's work, they have several drawbacks, as described below.

- All of the clients are always fuzzing the same set of seeds.
- They only parallelize the deterministic mutation part of AFL and fail to parallelize the deterministic mutation part.
- They synchronize the shared fuzzing status in a fixed time period but not immediately.
- They ignore to share the coverage information.

2.4. Data Races

In parallel computing, some uncontrolled accesses to shared data happen simultaneously, which results in race conditions [20].

In this paper, we focus on the specific race cases in parallel fuzzing. The key to handling this problem is to tackle the sharing objects appropriately. Through observation, we list some typical race cases:

2.4.1. Several Computing Nodes Access to the Same Seed

Accessing the same seed during the deterministic mutation phase produces massive repeated fuzzing, which is time-consuming.

2.4.2. Several Client Nodes Update the Coverage Information Together

As we mentioned in Section 2.2, the coverage information is stored in the bitmap. Bitmaps from different computing nodes with different coverage information reflect on different changed bytes. Merging these bitmaps without controlling, later-updated bitmaps may cover some valuable bits that others have updated before.

To solve these race cases and exceptions, we propose a set of strategies which is shown in Section 3.

3. Methodology

To improve the fuzzing efficiency and to make full use of computing resources, we design a parallel fuzzing framework P-fuzz. The overview of the P-fuzz framework is shown in Figure 1. We leverage database-centric architecture [21] that uses a database to handle the communication of computing nodes. The computing node deployed in the database is considered as a server. The database stores fuzzing status (including seeds and bitmap) to communicate with other computing nodes which are considered as clients. When new edges are covered by a client, the new coverage information and seeds are updated to the server immediately. To keep P-fuzz from unexpected situations, we set races and exceptions handling strategies in the server. For different target programs, we select different mutation strategies to fuzz.

Figure 1 also reveals the fuzzing process. First, each client gets a seed and the occupied seeds are marked by a flag. Second, each client starts fuzzing. When some interesting new edges are covered by a client, it uploads seeds and the bitmap. A service is responsible for merging the bitmap to avoid the data race. Then, other clients will receive the updated bitmaps.

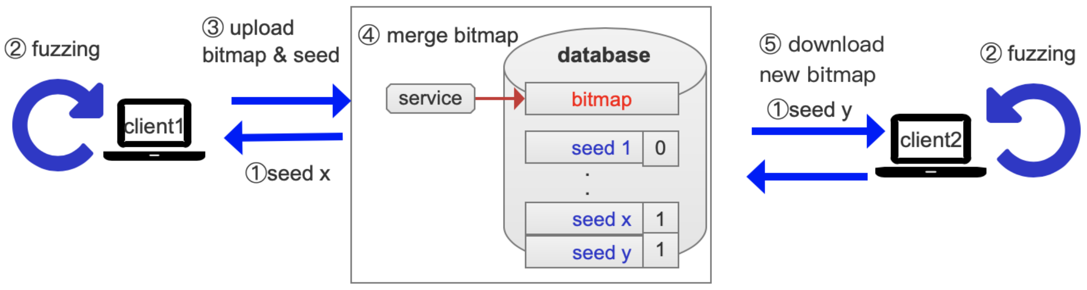


Figure 1. The overview of the P-fuzz framework.

3.1. *Dynamic Fuzzing Status Synchronizing and Workload-Balancing Mechanism*

Computing resources and fuzzing tasks are two entities of parallel fuzzing system. The most important work is to distribute fuzzing tasks to compute resources appropriately to achieve the balance. Previous studies [13,14] show us two drawbacks in tackling this work:

- Underutilizing the computing resources, which burdens the single core with many fuzzing tasks
- Sharing information (including seeds, queues, crashes, and hangs) with each client but not distributing them, which may lead to all computing nodes doing repeated work and not fully reflecting the advantages of parallelization. This case is depicted in Figure 2a.

We leverage database-centric [21] architecture to schedule workloads and synchronize the fuzzing status. A server with a database acts as the core of the whole system, and other computing nodes act as clients to communicate with the database. To make full use of computing resources and to enhance the fuzzing efficiency, we schedule the fuzzing tasks by letting each client fuzz different seeds (Figure 2b). To balance the workload, each client node will receive a new seed after completing fuzzing for a seed dynamically.

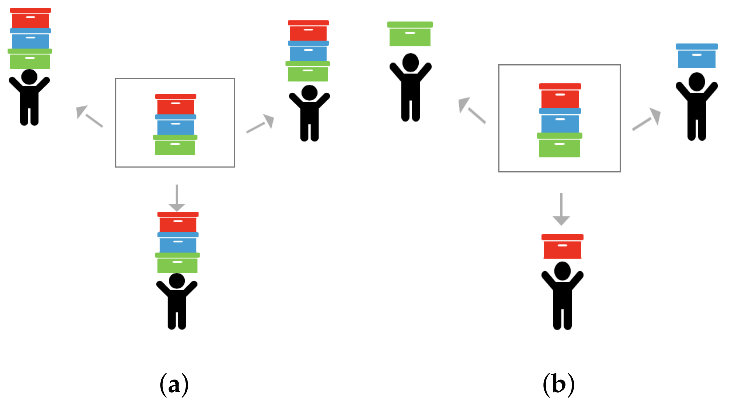


Figure 2. Examples of balancing workloads: (**a**) The example of distributing workloads by previous works. (**b**) The example of distributing workloads by P-fuzz.

We share seeds and the bitmap in the database. Also, we mark the sharing seeds with flags and time stamps in the database to identify whether this seed has been occupied by a client. Furthermore, we start a service to monitor the server which can solve the data race problem. In this way, all clients always work for valuable tasks by the scheduling mechanism of P-fuzz.

3.2. Races and Exceptions Handling Strategies

3.2.1. Case 1: Several Clients Access the Same Seed

As mentioned above, P-fuzz shares seeds produced by each client and stores them into a database. It schedules different clients to access different seeds to enhance the fuzzing efficiency. However, when several clients access a seed simultaneously, a data race happens.

To alleviate this situation, we set a flag attribute attached to the seed. The flag marks whether this seed is being fuzzed by a client. According to whether a seed is free or occupied, its flag is set to "0" or "1". A client checks the flag when it chooses seeds. If the flag of a seed is "1", the client will choose another seed to fuzz.

3.2.2. Case 2: Several Clients Update the Bitmap in the Database Together

We store the bitmap as a record in the database for sharing the coverage information. A data race happens as shown in Figure 3. Elements in the bitmap with "1" or "0" represent the edge uncovered or covered. The figure reveals that two clients update their new bitmaps (Figure 3b,c). If we do not control the merging process, to merge the bitmap of client1 and then client2, some valuable information will get lost as in Figure 3d.

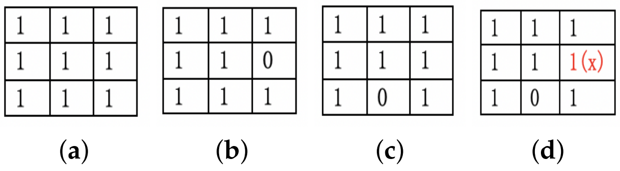


Figure 3. A race of updating bitmap from different clients: (**a**) the origin bitmap, (**b**) the bitmap updated from client1, (**c**) the bitmap updated from client2, and (**d**) the final bitmap (a "0" is covered).

To alleviate this situation, we start a service in the server to manage the merging operation. The service builds a queue to store the bitmap temporarily. When bitmaps from different clients come up to the database, they are enqueued according to the time order. The database merges these enqueued bitmaps by "AND(&)" operations one by one so that the bitmap maintains all the necessary information.

3.2.3. Case 3: A Client Quits Fuzzing Accidentally but it Does Not Finish a Complete Fuzzing Round

As mentioned above, we set a flag to mark whether the seed is occupied by a client. However, in parallel computing, a client sometimes quits with exceptions. At this time, the flag is "1" but the fuzzing process of the corresponding seed is not finished. Here, we consider that a complete fuzzing round is a seed which is fuzzed in a whole deterministic mutation process.

To solve this problem, we put a timestamp when the flag is set to "1". We also monitor if the fuzzing is overtime with the timestamp. This strategy assures exceptions will not disturb the parallel fuzzing.

3.3. Optimization

3.3.1. Immediate Response to Update

Different from Roving and the work of Martijn, which synchronize the sharing data in a fixed time period (such as 300 s in Roving), P-fuzz updates new seeds and bitmap data to the database when AFL produces them. The prompt action makes all clients in the system get updated seeds and the feedback information immediately.

3.3.2. The Selection of Mutation Strategies

According to the introduction in Section 2, nondeterministic and deterministic mutation strategies do well in different targets. Therefore, P-fuzz adopts both of them to fuzz. For most of the target programs, we set one client to do deterministic mutations and others to do nondeterministic mutations to cover more edges and to keep the efficiency of parallel fuzzing. For those target programs which are format aware, we set more clients to do deterministic mutations first to keep the format of files and less clients to do nondeterministic mutations. The quantity of clients to do which kind of mutation is determined by target programs.

4. Implementation

4.1. The Steps of Implementation

The steps of implementing the P-fuzz enviroment are shown below:

- Setting up and configuring the database in the server
- Configuring the service in the server
- Building a fuzzing environment (the AFL engine) in a Docker container
- Deploying the environment to all clients in a distributed system automatically
- Choosing a target program and starting fuzzing in each client node
- Each client updates new seeds and changes the bitmap during fuzzing
- Getting fuzzing results from the server

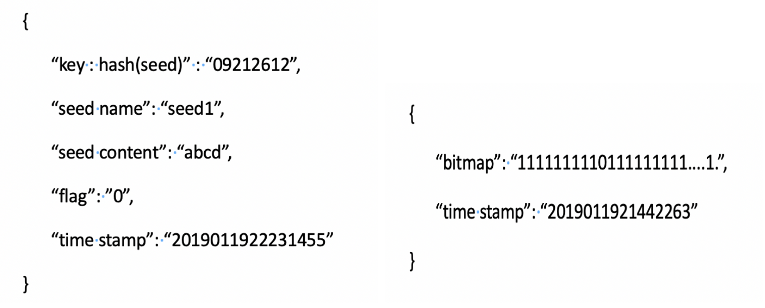
4.2. Server

The server is the core of the whole fuzzing system since the P-fuzz is based on the database-centric architecture. We deploy a MongoDB database on the server to store the sharing data.

4.2.1. Database

MongoDB [22] is an open-source document database, which is no-SQL with high performance, high availability, and automatic scaling. A collection in a database gathers a set of data in any type.

As shown in Figure 4a,b, we set two types of collections in the database. One is "seed". To avoid the situation that clients send seeds which have the same content, the first attribute is a hash value of the seed content. The second attribute records the name of the seed. The third attribute records the content of the seed. The fourth attribute is a flag to mark whether the seed is being fuzzed, and the last attribute is a time stamp, which is used to mark the time of a fuzzing start.



```
{
    "key : hash(seed)" : "09212612",
    "seed name": "seed1",
    "seed content": "abcd",
    "flag": "0",
    "time stamp": "2019011922231455"
}
```

```
{
    "bitmap": "1111111110111111111....1.",
    "time stamp": "2019011921442263"
}
```

(**a**) (**b**)

Figure 4. Two collections of database: (**a**) the seed collection in the database and (**b**) the bitmap collection in the database.

The other type of collection is "bitmap". In the whole database, there is only one bitmap collection because all clients need to share this bitmap to acquire the whole coverage information of the system. The first attribute in this collection is all bit information of the sharing bitmap. The second attribute is the time stamp to record the latest update time of the bitmap.

4.2.2. Service

As we discuss in Section 3, when several clients update the bitmap together, some bits in the bitmap will get lost. In order to solve this race, we start a service in the server to manage the merging operation. The service maintains a queue to store temporarily the coming bitmaps from clients according to the time order. Then, the service merges these bitmaps in the queue by an "*AND*" (&) operation.

4.3. Client

We choose a set of computers as clients. To build a fuzzing environment automatically, we utilize the ability of Docker. A Docker container is a lightweight package of software that includes everything needed to run an application [17]. We first configure a container with the AFL engine and its required environment. Based on the container, we use the Docker swarm to duplicate the fuzzing environment to all clients in the system.

At the start of fuzzing, each client downloads the chosen target program and a seed from the central database. When some interesting edges are found by a client, it updates these new seeds and the bitmap to the central database. Other clients will share the updated bitmap immediately. P-fuzz depends on this mechanism to share data in parallel fuzzing.

5. Experiment

5.1. Experiment Setup

We conduct experiments on a small-scale cluster which consists of eight desktops with Intel Core i7 3.4 GHz, 8 Core CPU, and 8 GB RAM running Ubuntu 16.04. In order to compare P-fuzz with another parallel fuzzing framework, we divide the eight desktops into two groups for testing two parallel fuzzing frameworks. We choose five programs in GNU Binutils [23] (nm, objdump, readelf, size, and strings), LAVA-M data set [24] (base64, md5sum, uniq, and who), two image processing tools (bmp2tiff and tiffinfo), and tcpdump as our target programs. Thus, we have 13 target programs to conduct experiments. We compare P-fuzz with AFL and a previous parallel fuzzing framework Roving for two hours. To prove the improvement in efficiency of P-fuzz, we record four indicators for each experiment:

- **Bitmap density.** This is an important indicator to measure the coverage of grey-box fuzzers. As Section 2.2 mentioned, a byte in the bitmap indicates an edge, which connects two or more basic blocks of the target program. The bitmap density indicates the ratio of changed bytes that the bitmap takes in the size of the bitmap.
- **Crashes.** This is the number of unique crashes that occur when executing the programs. Crashes are generated from test cases triggering target programs to produce a fatal signal (e.g.,SIGSEGV, SIGILL, and SIGABRT).
- **Speed.** We measure the speed by how many test cases have been executed per second (exe/s).
- **Inputs.** This is an indicator to calculate the quantity of seeds generated in the queue.

Before we start fuzzing, we need to compile target programs with AFL's compiler called *afl-gcc*. *afl-gcc* instruments the source code of targets and produces target binary files.

5.2. Results

To evaluate the efficiency, we take a 2-h rapid experiment on P-fuzz, AFL in a single node, and Roving to test the above indicators of nine target programs and LAVA-M data benchmarks. The result is listed in Table 1.

Table 1. Experiment results of three frameworks on nine target programs and LAVA-M data benchmarks.

Program	AFL				Roving				P-Fuzz			
	density	*inputs*	*crashes*	*speed*	*density*	*inputs*	*crashes*	*speed*	*density*	*inputs*	*crashes*	*speed*
nm	3.92	950	0	1089	6.57	2967	0	9928	6.76	5091	0	7351
strings	0.16	64	0	1022	0.16	395	0	4521	0.16	143	0	5011
objdump	8.48	1923	0	630	10.75	3777	0	12,352	12.49	7283	0	5612
size	3.54	605	0	2648	6.64	2907	0	11,204	6.15	8723	0	8051
readelf	7.27	1747	0	1219	12.1	6446	0	6252	9.24	1034	0	7006
tiffinfo	0.04	9	0	4001	0.04	10	0	15,903	4.81	754	0	8603
bmp2info	0.61	480	25	228	0.6	1826	26	2836	3.56	201	38	6488
tcpdump	3.61	775	0	1321	11.2	4072	0	5472	36.77	17,926	0	4812
nasm	8.34	2531	0	1337	10.28	2528	0	3362	10.56	9152	0	2914
base64	0.58	389	0	368	0.58	788	0	8345	1.15	678	2	5921
md5sum	0.83	156	0	206	1.01	2701	0	4366	0.86	412	0	688
uniq	0.36	57	0	783	0.36	188	1	3624	0.37	143	1	3200
who	2.46	178	0	1023	2.47	1008	1	6200	3.21	532	8	7722

As shown in the table, we can see P-fuzz covers more bytes of bitmap than AFL and Roving in most of target programs. The bitmap density of P-fuzz is 2.59× higher than AFL and 1.66× higher than Roving on average. Especially, the bitmap density reaches 36.77% in "tcpdump", which almost triples the bitmap density of Roving. It is worth mentioning that, in two image processing tools "tiffinfo" and "bmp2tiff", P-fuzz also shows its ability to handle format-awareness programs by utilizing deterministic mutation strategies. The bitmap density upper limits of both AFL and Roving in "tiffinfo" and "bmp2tiff" are 0.04% and 0.61%, while P-fuzz reaches 4.8% and 3.56%, respectively. However, the three frameworks get similar bitmap densities in "strings" and LAVA-M data benchmarks. The reason is that "strings" is a target program with fewer paths; all these three frameworks are easy to reach the covergence status. On the other hand, LAVA-M is a designed data set; parallel fuzzing without the improvement in the algorithm is hard to cover more edges.

Moreover, the rapid experiment in just 2 h is hard to find crashes. With the high-efficiency characteristic, P-fuzz speeds up the whole fuzzing process and finds more crashes than AFL and Roving in such a short time. In "who", P-fuzz triggers eight crashes while Roving only triggers one crash. Also, P-fuzz finds 38 crashes in "bmp2tiff", more than AFL's 25 crashes and Roving's 26 crashes.

The "speed" attribute in Table 1 is measured by the number of test cases executed per second. Because of parallelizing the fuzzing, P-fuzz easily gains an almost 4× speed up. However, the average speed is a little lower than the Roving. The reason is that Roving uses nondeterministic mutations in the whole fuzzing process, while P-fuzz combines the two mutation strategies.

The bitmap density measures the edge coverage of grey-box fuzzing. The increment rate of bitmap density also reflects the efficiency of tools. We select the bitmap density data of "objdump", which is shown in Figure 5, to prove the high efficiency of P-fuzz. The figure reveals the bitmap density increment during the start 1000 seconds of the experiment. P-fuzz surpasses AFL and Roving rapidly in five seconds and keeps increasing.

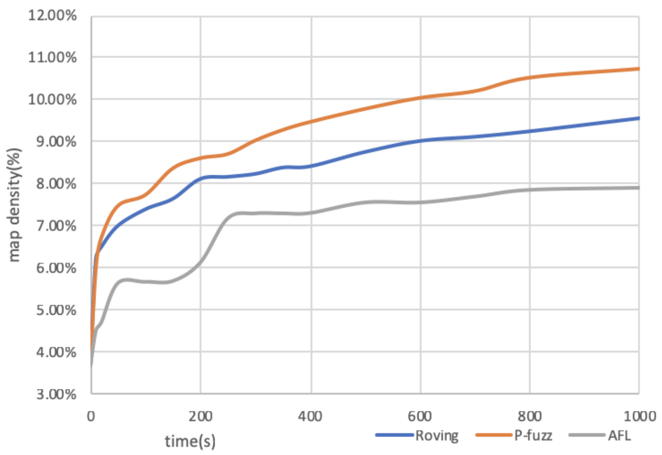


Figure 5. Comparison of bitmap density on objdump by AFL, Roving, and P-fuzz.

5.3. Analysis

As shown in Table 1, P-fuzz outperforms the other two framework. We try to analyze the strengths of P-fuzz.

5.3.1. P-Fuzz vs. AFL in a Single Node

The fuzzing efficiency of P-fuzz outperforms AFL. AFL in a single node is the baseline of experiments. We can see from the results the P-fuzz outperforms AFL by applying parallel computing technique. In the 2-h fuzzing, the edges P-fuzz covered and the paths produced are higher than that by AFL.

5.3.2. P-Fuzz vs. Roving

Roving does not share the feedback information of grey-box fuzzing. Roving shares test cases, queues, crashes, and hangs with each client in the system by synchronizing these fuzzing statuses to the server. However, the coverage information is also significant to fuzzing. P-fuzz uploads the bitmap as the coverage information to share edges that the whole framework has found with each client.

The mechanism of Roving takes up too much memory. The sharing mechanism of Roving is synchronizing all the test cases produced by four client nodes to the server, whether the test case is the same as with others. However, when fuzzing target programs which contain a large number of edges, the server of Roving is shut down because it does not support handling too many files. Compared with Roving, P-fuzz just uploads test cases as records to the database, which saves massive storage space compared to Roving.

The mutation strategy of Roving is monotonous. Roving only adopts nondeterministic mutation to make parallel fuzzing more randomly and rapidly. The executing speed of Roving is much higher than P-fuzz actually. However, the benefits of deterministic mutation are discarded, which leads to some complex programs being ignored by Roving.

6. Discussion

6.1. Advantage

Inheriting the effectiveness of AFL, P-fuzz improves the efficiency of grey-box fuzzing. The advantages of P-fuzz are discussed below:

1. **Utilizing the numerous computing resources to assist grey-box fuzzing.** Other works strive to improve the algorithm in a single computing node. These works actually enhance fuzzing in

different sides. Because of the orthogonality of the improvement of algorithm and computing resources, the advanced works can be applied in parallel fuzzing.

2. **Balancing the workload and sharing fuzzing status appropriately.** By distributing the workload to all client nodes in the system, P-fuzz makes full use of the computing resources. P-fuzz will not waste more time on edges which have been covered by getting bitmap information shared by all clients.
3. **Avoiding data races and exceptions the parallel fuzzing.** The data races and exceptions in parallel fuzzing influence not only the accuracy of results but also the efficiency of fuzzing. P-fuzz focuses on some typical cases and adopts valid strategies to avoid them.

6.2. Limitations

Although P-fuzz enhances the efficiency of AFL and outperforms the parallel fuzzing framework Roving, still a limitation exists in this work. For some tiny target programs, all edges can be covered rapidly. It is not worthy to use too many hardware overheads to exchange a little improvement in efficiency. We should try to find a balance to make a trade-off between the overhead of hardware resources and efficiency.

6.3. Future Work

In our further work, we will enhance the efficiency and effectiveness of P-fuzz in two directions. One is incorporating the advanced algorithm of fuzzing. Because of the orthogonality of parallel fuzzing optimization and algorithm improvement, we can apply an improved algorithm of AFL or some other techniques such as concolic execution [25] in P-fuzz.

The other direction is putting the parallel fuzzing into a large-scale cluster. In this case, the ability of database interaction may become the bottleneck of parallel fuzzing. Also, the data race in fuzzing will occur more frequently. However, we should focus on the possibility of significantly improving the efficiency by gathering the power of massive computing nodes. Therefore, how to tackle these bottlenecks is what we should strive for.

7. Related Work

7.1. Fuzzing Tools

Fuzzing tools can be classified into three types based on the knowledge and information acquired from the source code of a target program: they are white-box, black-box, and grey-box fuzzer.

7.1.1. White-Box Fuzzing

The white-box fuzzer has full knowledge of the source code (e.g., internal logic and structure) and uses the control structure of the procedural design to derive test cases. Current white-box fuzzing tools contains Sage [26], Angr [27], and KLEE [28], etc.

7.1.2. Black-Box Fuzzing

The black-box fuzzer does not have any knowledge of the source code, but it generates test cases randomly and swiftly. Some typical fuzzers such as Radamsa [29], zzuf [30], and Peachfuzz [31] did remarkable work in this field. Peachfuzz [31] have the ability to fuzz programs which are format awaren by providing description files.

7.1.3. Grey-Box Fuzzing

The grey-box fuzzer tries to combine the efficiency and effectiveness of black-box fuzzers and white-box fuzzers, which masters limited knowledge of the internal working of the target program. Through collecting the feedback information of target programs, grey-box fuzzers show the competitiveness of mutating test cases with valid guidance. It is implemented by lightweight instrumentation or other mechanisms to get the feedback of program executions, such as code coverage for the fuzzing process. AFL [4] is a state-of-the-art grey-box fuzzer of which the principles are speed, reliability, and ease of use. AFL instruments then compiled program to get the edge coverage information. Bohme et al. designed AFLfast, which intended to fuzz edges covered with low-frequency. Gan et al. introduced CollAFL [6], which mitigated the collision of bitmap data structure by providing more accurate coverage information. Bohme also implemented a directed grey-box fuzzing tool, AFLGo [7], towards the dangerous locations which tended to produce vulnerabilities. Zhang [5] et al. leveraged hardware mechanism (Intel processor trace) to collect edge information and fed this information back to the fuzzing process. All of these extensions gained higher coverage and found more bugs than AFL. However, these works based on improving algorithms are still compute-intensive and the efficiency is limited.

7.2. Other Fuzzing Tools Based on Parallel Technique

Some previous works try to leverage parallel computing technique to speed up the fuzzing process. The technique collects a group of computing resources to decompose heavy fuzzing tasks.

To enhance the efficiency of symbolic execution, Cloud9 [32] shares the searching scope into some pieces and each computing node shares the workload. Liang [12] also solved the challenge of path explosion by putting results into different computing nodes; this method is similar to our mechanism of distributing seeds.

For the parallel coverage-based grey-box fuzzing, more attention is paid to distributing the fuzzing test cases to balance the system workload. Xie [11] used grid computing for large-scale fuzzing in 2010, which reduces almost two-thirds of fuzzing time. It was implemented by dividing fuzzing jobs into tasks, by storing them in a server, and by scheduling remote clients to download them. ClusterFuzz [33] is a scalable fuzzing infrastructure which supports coverage-based grey-box fuzzing (e.g., libFuzzer and AFL) and black-box fuzzing. It is used by Google for fuzzing the Chrome browser and serves as the fuzzing backend for OSS-Fuzz[34].

8. Conclusions

In this paper, we leverage the parallel computing technique to improve the efficiency of grey-box fuzzing, which is different from traditional developing fuzzing algorithms. We implement the parallel fuzzing framework P-fuzz by applying the database-centric architecture, which consists of a database server and several clients. P-fuzz balances the workload by giving different clients different seeds. Also, it shares seeds and the bitmap data with each client to synchronize the fuzzing status. Futhermore, P-fuzz handles data races and exceptions in the parallel fuzzing. For fuzzing different types of targets, P-fuzz selects appropriate mutation strategies.

Finally, we conduct experiments to compare P-fuzz with AFL in a single node and a parallel fuzzing framework Roving in nine target programs and LAVA-M data benchmarks. The experimental result proves that P-fuzz improves the efficiency of the grey-box fuzzer. From the result, we believe that the use of computing resources in software testing is a worthwhile exploration and should be a widely used idea.

Author Contributions: conceptualization, X.Z.; methodology, X.H., Q.Y. and C.S.; software, Q.Y. and H.Z.; validation, C.S. and Q.Y.; investigation, C.S.; resources, X.Z.; writing—original draft preparation, C.S.; writing—review and editing, C.S.; supervision, X.Z. and K.L.; project administration, X.Z.;

Funding: This work is supported by Tianhe Supercomputer Project 2018YFB0204301.

Acknowledgments: We sincerely thank the reviewers for your insightful comments that help us improve this work.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Sutton, M.; Greene, A.; Amini, P. *Fuzzing: Brute Force Vulnerability Discovery*; Pearson Education: New York, NY, USA, 2007.
2. Cve List. Available online: http://cve.mitre.org/ (accessed on 1 January 1988).
3. Lu, K.; Wang, P.-F.; Li, G.; Zhou, X. Untrusted hardware causes double-fetch problems in the i/o memory. *J. Comput. Sci. Technol.* **2018**, *33*, 587–602.
4. Zalewski, M. American Fuzzy Lop. Available online: http://lcamtuf.coredump.cx/afl/ (accessed on 6 February 2015).
5. Zhang, G.; Zhou, X.; Luo, Y.; Wu, X.; Min, E. Ptfuzz: Guided fuzzing with processor trace feedback. *IEEE Access* **2018**, *6*, 37302–37313.
6. Gan, S.; Zhang, C.; Qin, X.; Tu, X.; Li, K.; Pei, Z.; Chen, Z. Collafl: Path sensitive fuzzing. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–24 May 2018; pp. 679–696.
7. Böhme, M.; Pham, V.-T.; Nguyen, M.-D.; Roychoudhury, A. Directed greybox fuzzing. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October– 3 November 2017; pp. 2329–2344.
8. Wilson, G.V. The History of the Development of Parallel Computing. 1994. Available online: http://ei.cs.vt.edu/history/Parallel.html (accessed on 28 Oct. 1994).
9. Almasi, G.S.; Gottlieb, A. *Highly Parallel Computing*; U.S. Department of Energy: Washington, DC, USA, 1988.
10. Amazon Spot Instance. Available online: https://aws.amazon.com/ec2/spot/ (accessed on 2 January 2013).
11. Yan, X. Using Grid Computing for Large Scale Fuzzing. Ph.D. Thesis, Universidade Nova de Lisboa, Lisbon, Portugal, 2010.
12. Liang, H.; Xiaoyu, Y.; Yu, D.; Zhang, P.; Shuchang, L. Parallel smart fuzzing test. *J. Tsinghua Univ. (Sci. Technol.)* **2015**, *54*, 14–19.
13. Roving. Available online: https://github.com/richo/Roving (accessed on 21 July 2015).
14. Distributed Fuzzing for afl. Available online: https://github.com/richo/Roving/ (accessed 15 June 2015).
15. Liang, J.; Jiang, Y.; Chen, Y.; Wang, M.; Zhou, C.; Sun, J. Pafl: Extend fuzzing optimizations of single mode to industrial parallel mode. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Lake Buena Vista, FL, USA, 4–9 November 2018; pp. 809–814.
16. Chen, Y.; Jiang, Y.; Ma, F.; Liang, J.; Wang, M.; Zhou, C.; Su, Z.; Jiao, X. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. *arXiv* **2018**, arXiv:1807.00182.
17. Docker. Available online: https://www.docker.com/ (accessed on 14 March 2008).
18. Khan, M.E.; Khan, F. A comparative study of white box, black box and grey box testing techniques. *Int. J. Adv. Comput. Sci. Appl.* **2012**, *3*, doi:10.14569/IJACSA.2012.030603.
19. American Fuzzy Lop (AFL) Fuzzer-Technical Details. Available online: http://lcamtuf.coredump.cx/afl/technical_details.txt (accessed on 6 Feb. 2015).
20. Corbet, J.; Rubini, A.; Kroah-Hartman, G. *Linux Device Drivers: Where the Kernel Meets the Hardware*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2005.
21. Database-Centric Architecture. Available online: https://en.wikipedia.org/wiki/Database-centric_architecture (accessed on 12 Mar. 2001).
22. Mongodb. Available online: https://www.mongodb.com/ (accessed on 28 April 2000).
23. GNU Binutils. Available online: http://www.gnu.org/software/binutils/ (accessed on 9 September 1988).
24. Dolan-Gavitt, B.; Hulin, P.; Kirda, E.; Leek, T.; Mambretti, A.; Robertson, W.; Ulrich, F.; Whelan, R. Lava: Large-scale automated vulnerability addition. In Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2016; pp. 110–121.
25. Baldoni, R.; Coppa, E.; D'Elia, D.C.; Demetrescu, C.; Finocchi, I. A survey of symbolic execution techniques. *ACM Comput. Surv.* **2016**, *51*, 1–39.
26. Godefroid, P.; Levin, M.Y.; Molnar, D. Sage: Whitebox fuzzing for security testing. *Commun. ACM* **2012**, *55*, 40–44.
27. Angr. Available online: https://angr.io/ (accessed on 15 September 2014).

28. Cadar, C.; Dunbar, D.; Engler, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Proceedings of the Usenix Conference on Operating Systems Design & Implementation, San Diego, CA, USA, 8–10 December 2009.
29. Radamsa. Available online: https://github.com/aoh/radamsa (accessed on 9 August 2011).
30. Zzuf. Available online: http://caca.zoy.org/wiki/zzuf (accessed on 30 May 2015).
31. Peach. Available online: https://www.peach.tech (accessed on 1 April 2008).
32. Ciortea, L.; Zamfir, C.; Bucur, S.; Chipounov, V.; Candea, G. Cloud9: A software testing service. *ACM SIGOPS Oper. Syst. Rev.* **2010**, *43*, 5–10.
33. ClusterFuzz. Available online: https://google.github.io/clusterfuzz/ (accessed on 1 December 2016).
34. Serebryany, K. *Oss-Fuzz-Google's Continuous Fuzzing Service for Open Source Software*; USENIX: Berkeley, CA, USA, 2017.