


Article

# P-fuzz: a parallel grey-box fuzzing framework

Congxi Song <sup>1</sup> , Xu Zhou <sup>1\*</sup>, Qidi Yin<sup>1</sup>, Xinglu He <sup>1</sup>, Hangwei Zhang <sup>1</sup> and Kai Lu<sup>1</sup>

<sup>1</sup> College of Computer, National University of Defense Technology, Changsha 410073, China; congxi1994@sohu.com

\* Correspondence: zhouxu@nudt.edu.cn

Version November 15, 2019 submitted to Journal Not Specified

**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 2.59X and Roving about 1.66X 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 monitoring for exceptions[2]. In this way, thousands of security vulnerabilities are discovered by 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. A state-of-the-art grey-box fuzzer American Fuzzy Lop (AFL)[6] 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, 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[1][6][8]-[10]. Unlike all those works, we consider this efficiency problem of grey-box fuzzing in a different point of view. Instead of being limited to improve 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[20][21]. We can get massive cheap computing resource easily (e.g. by using the Amazon spot instance[11]). 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. Besides, 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[12] 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.[13] presented a distributed fuzzing framework which can manage

36 computing resources in a cluster and schedule resources to many submitted fuzzing jobs. However, it  
37 does not intend to accelerate a single fuzzing job. Roving[14] and the work of Martijn[15] can parallel  
38 AFL to fuzz a single program with distributed computing nodes. However, they only parallelize the  
39 non-deterministic mutation part of AFL and fail to parallelize the deterministic mutation part. Also,  
40 they do not synchronize the coverage information which is a crucial part of grey-box fuzzers. PAFL[35]  
41 and Enfuzz[34] can parallelize several typical fuzzers together, which are proposed in 2018 and 2019.  
42 These two works are characterized by parallelizing diverse fuzzing tools to solve problems with the  
43 characteristics of different tools. However, these framework can not share and synchronize tasks and  
44 resources across machines.

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

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

51 We implement a parallel grey-box fuzzing framework P-fuzz to solve these questions. P-fuzz  
52 consists of computing nodes to accelerate fuzzing. To share seeds and the coverage information which  
53 is stored in the bitmap, we leverage a key-value database. A computing node fetches a seed from  
54 the database and begins its fuzzing process: Firstly, the node mutates the seed to generate test cases.  
55 Then the node sends test cases to the target program and monitors the execution of the target. When  
56 the node hits uncovered edges, it adds the corresponding test case to the database as a new seed and  
57 feedbacks this updated coverage information to the database to share benefits with other computing  
58 nodes. Also, we apply strategies to dynamically distribute fuzzing tasks to different nodes to achieve  
59 balancing the workload. To handle data races and exceptions, we analyze a set of specific cases and  
60 propose solutions for each case. In addition, we use Docker[16] technique to build the environment  
61 of fuzzing framework and copy this environment to all computing nodes in the distributed system  
62 automatically. P-fuzz is capable of parallelizing fuzzing tools across machines, which is different  
63 from other frameworks just sharing fuzzing status in a file system. Also, it provides the scalability of  
64 parallelizing various fuzzing tools.

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

69 There are four contributions in this work:

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

## 76 2. Background

### 77 2.1. The classification of fuzzing

78 According to the knowledge and information acquired from the target programs, fuzzing can be  
79 divided into white-box, black-box, and grey-box fuzzing[33]. The white-box fuzzer has full knowledge  
80 of the source code (e.g. internal logic and data structures) and uses the control structure of the  
81 procedural design to derive test cases. In contrast, the black-box fuzzer does not have any knowledge  
82 of the target program, thus it generates test cases randomly and swiftly. The grey-box fuzzer combines

83 the efficiency and effectiveness of black-box fuzzers and white-box fuzzers, which masters limited  
 84 knowledge of target programs. Currently, the grey-box fuzzing technique is practical and widely used  
 85 in the software testing and vulnerability detection as it is lightweight, fast and easy-to-use[1].

86 The grey-box fuzzing process usually contains three steps:

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

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

## 94 2.2. The details about AFL

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

### 98 2.2.1. Seed

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

### 102 2.2.2. Bitmap

103 Bitmap is a data structure which stores the coverage information of fuzzing. The bitmap size  
 104 of AFL is 64 Kilobytes. A byte in the bitmap indicates an edge, which connects two or more basic  
 105 blocks of the target program. The eight bits in a byte describe how many times this edge is covered.  
 106 We use a tuple to express a edge, for example, there are basic block  $A$  and  $B$ , then a  $tuple(A, B)$  means a  
 107 edge from previous basic block  $A$  to current basic block  $B$ . If a test case covers a new edge, the bitmap  
 108 will record this changed coverage information by updating the corresponding byte. A mechanism to  
 109 index the bitmap is shown as Eq.1. By simply reading the bitmap, AFL knows whether a edge is new  
 110 covered or not and decides to store or discard a test case[1].

$$(A \oplus B) \% BITMAP\_SIZE \quad (1)$$

111 Moreover, AFL runs deterministic mutations and non-deterministic mutations. Deterministic  
 112 mutation strategies produce test cases and small diffs between the non-crashing and crashing  
 113 inputs[24]. Non-deterministic mutation strategies can make fuzzing achieve high coverage rapidly by  
 114 random combining deterministic strategies. Roving[14] relies on the non-determinism of AFL to cover  
 115 more edges faster. However, for fuzzing a target program whose input files are in a complex format,  
 116 random mutations will destroy the format of files. Therefore, utilizing appropriate strategies to fuzz  
 117 different programs is necessary.

## 118 2.3. The discussion of parallel mechanism in fuzzing

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

124 Two fuzzing frameworks extend the parallel mechanism in AFL. One is Roving[14], which is  
 125 implemented by running multiple copies of AFL on multiple computing nodes, all of them fuzzing the

126 same target. It benefits from the client-server structure which shares crashes, hangs, and queues of  
127 each client. Each computing node plays a role in a client or server. Every 300 seconds, clients update  
128 the fuzzing environment by uploading and downloading changes. The whole framework is scheduled  
129 by the central server. The other is the work of Martijn[15]. The main idea of this work is approximate  
130 to Roving, and the difference between them is the implementation.

131 Although the two frameworks utilize computing resources and parallelize the fuzzing progress,  
132 which makes each client benefits from each other's work. They have several drawbacks as below.

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

#### 138 2.4. Data races

139 In parallel computing, some uncontrolled accesses to shared data happen simultaneously, which  
140 results in race conditions[22].

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

##### 144 2.4.1. Several computing nodes access to the same seed

145 Accessing to the same seed during deterministic mutation phase produce massive repeated  
146 fuzzing, which is time-consuming.

##### 147 2.4.2. Several client nodes update the coverage information together

148 As we mentioned in section 2.2, the coverage information is stored in the bitmap. Bitmaps from  
149 different computing nodes with different coverage information, which reflects on different changed  
150 bytes. Merging these bitmaps without controlling, later updated bitmap may cover some valuable bits  
151 others have updated before.

152 To solve these race cases and exceptions, we propose a set of strategies which is shown in section  
153 3.

### 154 3. Methodology

155 To improve the fuzzing efficiency and make full use of computing resources, we design a parallel  
156 fuzzing framework P-fuzz. The overview of P-fuzz framework is shown in Fig.1. We leverage  
157 Database-centric architecture[23] that using a database to handle the communication of computing  
158 nodes. The computing node deployed the database is considered as a server. The database stores  
159 fuzzing status(including seeds and bitmap) to communicate with other computing nodes which are  
160 considered as clients. When new edges are covered by a client, the new coverage information and  
161 seeds are updated to the server immediately. To keep P-fuzz from unexpected situations, we set races  
162 and exceptions handling strategies in the server. For different target programs, we select different  
163 mutation strategies to fuzz.

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

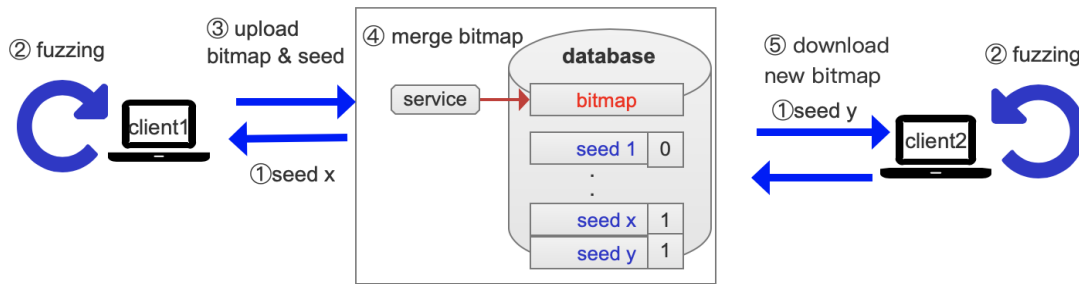


Figure 1. The overview of P-fuzz framework

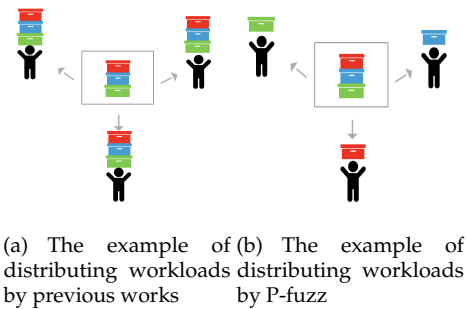


Figure 2. Examples of balancing workloads

### 168 3.1. Dynamic fuzzing status synchronizing and workload balancing mechanism

169 Computing resources and fuzzing tasks are two entities of parallel fuzzing system. And the most  
 170 important work is to distribute fuzzing tasks to computing resources appropriately to achieve the  
 171 balance. Previous studies[14][15] show us two drawbacks in tackling this work:

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

176 We leverage Database-centric[23] architecture to schedule workloads and synchronize the fuzzing  
 177 status. A server with a database acts as the core of the whole system, and other computing nodes act  
 178 as clients to communicate with the database. To make full use of computing resources and enhance the  
 179 fuzzing efficiency, we schedule the fuzzing tasks by letting each client fuzz different seeds(Fig.2(b)).  
 180 To balance the workload, each client node will receive a new seed after completing fuzzing a seed  
 181 dynamically.

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

### 186 3.2. Races and exceptions handling strategies

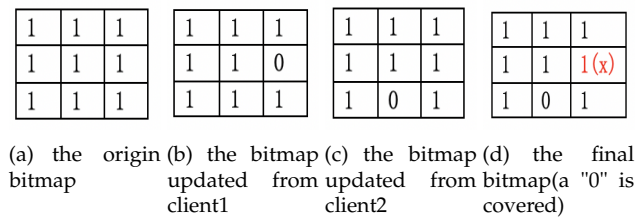
#### 187 3.2.1. Case 1: several clients access to the same seed.

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

191 To alleviate this situation, we set a flag attribute attaches to the seed. The flag marks whether this  
 192 seed is being fuzzed by a client. According to whether a seed is free or occupied, its flag is set to "0" or  
 193 "1". A client checks the flag when it chooses seeds. If the flag of a seed is "1", the client will choose  
 194 other seeds to fuzz.

195 3.2.2. Case 2: several clients update the bitmap in the database together.

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



**Figure 3.** A race of updating bitmap from different clients

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

205 3.2.3. Case 3: a client quits fuzzing accidentally but it does not finish a complete fuzzing round.

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

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

### 213 3.3. Optimization

#### 214 3.3.1. Immediate response to update

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

#### 219 3.3.2. The selection of mutation strategies

220 According to the introduction in section 2, non-deterministic and deterministic mutation strategies  
 221 do well in different targets. Therefore, P-fuzz adopts both of them to fuzz. For most of the target  
 222 programs, we set one client to do deterministic mutations and others to do non-deterministic mutations  
 223 to cover more edges and keep the efficiency of parallel fuzzing. For those target programs which are  
 224 format-awareness, we set more clients to do deterministic mutations first to keep the format of files,

225 and less clients do non-deterministic mutations. The quantity of clients to do which kind of mutation  
 226 is determined by target programs.

## 227 4. Implementation

### 228 4.1. The steps of implementation

229 The steps of implementing the P-fuzz environment is shown below:

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

### 237 4.2. Server

238 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.

```

  {
    "key": "hash(seed)": "09212612",
    "seed name": "seed1",
    "seed content": "abcd",
    "flag": "0",
    "time stamp": "2019011922231455"
  }
  {
    "bitmap": "1111111101111111....1",
    "time stamp": "2019011921442263"
  }
  
```

(a) the seed collection in the database      (b) the bitmap collection in the database

**Figure 4.** Two collections of database

239

#### 240 4.2.1. Database

241 MongoDB[32] is an open-source document database, which is no-SQL with high performance,  
 242 high availability and automatic scaling. A collection in a database gathers a set of data in any types.

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

248 The other type of collection is "bitmap". In the whole database, there is only one bitmap collection,  
 249 because all clients need to share this bitmap to acquire the whole coverage information of the system.  
 250 The first attribute in this collection is all bits information of the sharing bitmap. And the second  
 251 attribute is the timestamp to record the latest updating time of bitmap.

#### 252 4.2.2. Service

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

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

Program	AFL				Roving				P-fuzz			
	<i>density</i>	<i>inputs</i>	<i>crashes</i>	<i>speed</i>	<i>density</i>	<i>inputs</i>	<i>crashes</i>	<i>speed</i>	<i>density</i>	<i>inputs</i>	<i>crashes</i>	<i>speed</i>
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	12352	12.49	7283	0	5612
size	3.54	605	0	2648	6.64	2907	0	11204	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	15903	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	17926	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

### 257 4.3. Client

258 We choose a set of computers as clients. To build a fuzzing environment automatically, we utilize  
 259 the ability of Docker. A Docker container is a lightweight package of software that includes everything  
 260 needed to run an application[16]. We first configure a container with the AFL engine and its required  
 261 environment. Based on the container, we use the Docker swarm to duplicate the fuzzing environment  
 262 to all clients in the system.

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

## 267 5. Experiment

### 268 5.1. Experiment setup

269 We conduct experiments on a small-scale cluster which consists of eight desktops with Intel Core  
 270 i7 3.4GHz 8 Core CPU and 8GB RAM running Ubuntu 16.04. In order to compare P-fuzz with another  
 271 parallel fuzzing framework, we divide the eight desktops into two groups for testing two parallel  
 272 fuzzing frameworks. We choose five programs in GNU Binutils[17](nm, objdump, readelf, size and  
 273 strings), LAVA-M data set[18](base64, md5sum, uniq, and who), two image processing tools(bmp2tiff  
 274 and tiffinfo), and tcpdump as our target programs. Thus, we have 13 target programs to conduct  
 275 experiments. We compare P-fuzz with AFL and a previous parallel fuzzing framework Roving for two  
 276 hours. To prove the improvement in efficiency of P-fuzz, we record four indicators for each experiment:

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

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



## 288 5.2. Results

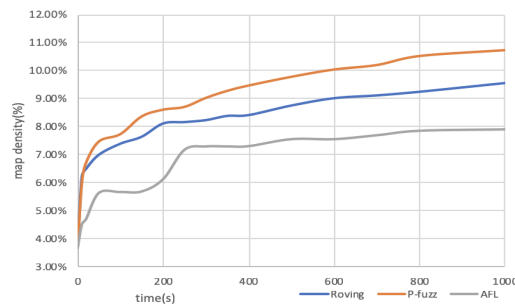
289 To evaluate the efficiency, we take a 2-hour rapid experiment on P-fuzz, AFL in a single node, and  
 290 Roving to test the above indicators of nine target programs and LAVA-M data benchmarks. The result  
 291 is listed in Table 1.

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

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

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

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



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

## 316 5.3. Analysis

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

### 319 5.3.1. P-fuzz vs. AFL in a single node:

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

### 324 5.3.2. P-fuzz vs. Roving:

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

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

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

## 339 6. Discussion

### 340 6.1. Advantage

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

- 343 1. **Utilizing the numerous computing resources to assist grey-box fuzzing.** Other work strives to  
344 improve the algorithm in a single computing node. These works actually enhance fuzzing in  
345 different sides. Because of the orthogonality of the improvement of algorithm and computing  
346 resources, the advanced works can be applied in parallel fuzzing.
- 347 2. **Balancing the workload and sharing fuzzing status appropriately.** By distributing the  
348 workload to all client nodes in the system, P-fuzz makes full use of the computing resources.  
349 P-fuzz will not waste more time to edges which have covered by getting the bitmap information  
350 shared by all clients.
- 351 3. **Avoiding data races and exceptions the parallel fuzzing.** The data races and exceptions in  
352 parallel fuzzing influence not only the accuracy of results but also the efficiency of fuzzing.  
353 P-fuzz focuses on some typical cases and adopts valid strategies to avoid them.

### 354 6.2. Limitation

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

### 360 6.3. Future work

361 In our further work, we will enhance the efficiency and effectiveness of P-fuzz in two direction.  
362 One is incorporating the advanced algorithm of fuzzing. Because of the orthogonality of parallel  
363 fuzzing optimization and algorithm improvement, we can apply an improved algorithm of AFL or  
364 some other techniques such as concolic execution[31] in P-fuzz.

365 The other direction is putting the parallel fuzzing into a large-scale cluster. In this case, the ability  
366 of database interaction may become the bottleneck of parallel fuzzing. Also, the data race in fuzzing  
367 will occur more frequently. However, we should focus on the possibility of significantly improving

368 the efficiency by gathering the power of massive computing nodes. Therefore, how to tackle these  
369 bottlenecks is we should strive for.

## 370 7. Related work

### 371 7.1. Fuzzing tools

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

#### 374 7.1.1. White-box fuzzing

375 The white-box fuzzer has full knowledge of source code (eg. internal logic and structure) and  
376 uses the control structure of the procedural design to derive test cases. Current white-box fuzzing  
377 tools contains Sage[3], Angr[25] and KLEE[26] etc.

#### 378 7.1.2. Black-box fuzzing

379 The black-box fuzzer does not have any knowledge of source code but it generates test cases  
380 randomly and swiftly. Some typical fuzzers such as Radamsa[27], zzuf[28] and Peachfuzz[29] which  
381 did remarkable work in this field. Peachfuzz[29] have the ability to fuzz programs which are  
382 format-awareness by providing description files.

#### 383 7.1.3. Grey-box fuzzing

384 The grey-box fuzzer try to combine the efficiency and effectiveness of black-box fuzzers  
385 and white-box fuzzers, which masters limited knowledge of the internal working of the target  
386 program. Through collecting the feedback information of target programs, grey-box fuzzers show  
387 the competitiveness of mutating test cases with the valid guidance. It is implemented by lightweight  
388 instrumentation or other mechanisms to get the feedback of program executions, such as code  
389 coverage for the fuzzing process. AFL[6] is a state-of-the-art grey-box fuzzer whose principles  
390 are speed, reliability, and ease of use. AFL instruments the compiled program to get the edge  
391 coverage information. Bohme et al. designed AFLfast[8] which intended to fuzz edges covered with  
392 low-frequency. Gan et al introduced CollAFL[9], which mitigated the collision of bitmap data structure  
393 by providing more accurate coverage information. Bohme also implemented a directed grey-box  
394 fuzzing tool AFLGo[10] towards the dangerous locations which tended to produce vulnerabilities.  
395 Zhang[1] et al. leveraged hardware mechanism (Intel Processor Trace) to collect edge information,  
396 and fed this information back to the fuzzing process. All of these extensions gained higher coverage  
397 and found more bugs than AFL. However, these work based on improving algorithms are still  
398 compute-intensive and the efficiency is limited.

### 399 7.2. Other fuzzing tools based on parallel technique

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

402 To enhance the efficiency of symbolic execution, Cloud9[5] shares the searching scope into  
403 some pieces, each computing node shares the workload. Liang[13] also solved the challenge of path  
404 explosion by putting results into different computing nodes, this method is similar to our mechanism  
405 of distributing seeds.

406 For the parallel coverage-based grey-box fuzzing, more attention is paid to distribute the fuzzing  
407 test cases to balance the system workload. Xie[12] used grid computing for large scale fuzzing in 2010,  
408 which reduce almost two-thirds of fuzzing time. It was implemented by dividing fuzzing jobs into  
409 tasks, storing them in a server and scheduling remote clients to download them. ClusterFuzz[30] is a  
410 scalable fuzzing infrastructure which supports for coverage-based grey-box fuzzing (e.g. libFuzzer

411 and AFL) and black-box fuzzing. It is used by Google for fuzzing the Chrome browser and serves as  
412 the fuzzing backend for OSS-Fuzz.

## 413 8. Conclusion

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

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

## 426 Acknowledgment

427 We sincerely thank reviewers for your insightful comments that help us improve this work.

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

## 429 References

- 430 1. G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "Ptfuzz: Guided fuzzing with processor trace feedback," *IEEE*  
431 *Access*, vol. 6, pp. 37302–37313, 2018.
- 432 2. M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.
- 433 3. P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Communications of*  
434 *the ACM*, vol. 55, no. 3, pp. 40–44, 2012.
- 435 4. K. Serebryany, "Oss-fuzz-google's continuous fuzzing service for open source software," 2017.
- 436 5. L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, "Cloud9: A software testing service," *ACM*  
437 *SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 5–10, 2010.
- 438 6. M. Zalewski, "American fuzzy lop," [Online]. Available: <http://lcamtuf.coredump.cx/afl/>
- 439 7. "Cve list," [Online]. Available: <http://cve.mitre.org/>.
- 440 8. M. Böhme, "Aflfast. new," 2017.
- 441 9. S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE*  
442 *Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 679–696.
- 443 10. M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of*  
444 *the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2329–2344.
- 445 11. "Amazon spot instance," [Online]. Available: <https://aws.amazon.com/ec2/spot/>.
- 446 12. X. Yan, "Using grid computing for large scale fuzzing," Ph.D. dissertation, Lisbon: Universidade Nova de  
447 Lisboa, 2010.
- 448 13. H. LIANG, Y. Xiaoyu, D. Yu, P. ZHANG, and L. Shuchang, "Parallel smart fuzzing test," *Journal of Tsinghua*  
449 *University (Science and Technology)*, vol. 54, no. 1, pp. 14–19, 2015.
- 450 14. "Roving," [Online]. Available: <https://github.com/riche/Roving>.
- 451 15. "Distributed fuzzing for afl," [Online]. Available: <https://github.com/riche/Roving/>.
- 452 16. "Docker," [Online]. Available: <https://www.docker.com/>.
- 453 17. "GNU Binutils," [Online]. Available: <http://www.gnu.org/software/binutils/>
- 454 18. B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava:  
455 Large-scale automated vulnerability addition," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE,  
456 2016, pp. 110–121.
- 457 19. K. Lu, P.-F. Wang, G. Li, and X. Zhou, "Untrusted hardware causes double-fetch problems in the i/o memory,"  
458 *Journal of Computer Science and Technology*, vol. 33, no. 3, pp. 587–602, 2018.

- 459 20. G. V. Wilson, "The history of the development of parallel computing," URL: <http://ei.cs.vt.edu/history/Parallel.html>, 1994.
- 460
- 461 21. G. S. Almasi and A. Gottlieb, "Highly parallel computing," 1988.
- 462 22. J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers: Where the Kernel Meets the Hardware*. "O'Reilly Media, Inc.", 2005.
- 463
- 464 23. "database-centric architecture," [Online]. Available: [https://en.wikipedia.org/wiki/Database-centric\\_architecture](https://en.wikipedia.org/wiki/Database-centric_architecture)
- 465 24. "American Fuzzy Lop (AFL) Fuzzer-Technical Details" [Online]. Available: [http://lcamtuf.coredump.cx/afl/technical\\_details.txt](http://lcamtuf.coredump.cx/afl/technical_details.txt)
- 466
- 467 25. "Angr," [Online]. Available: <https://angr.io/>
- 468 26. C. Cadar, D. Dunbar, and D. Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Usenix Conference on Operating Systems Design & Implementation*, 2009.
- 469
- 470 27. "radamsa," [Online]. Available: <https://github.com/aoh/radamsa>
- 471 28. "zzuf," [Online]. Available: <http://caca.zoy.org/wiki/zzuf>
- 472 29. "peach," [Online]. Available: <https://www.peach.tech>
- 473 30. "ClusterFuzz," [Online]. Available: <https://google.github.io/clusterfuzz/>
- 474 31. R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *Acm Computing Surveys*, vol. 51, no. 3, pp. 1–39, 2016.
- 475
- 476 32. "Mongodb" [Online]. Available: <https://www.mongodb.com/>
- 477 33. M. E. Khan, F. Khan *et al.*, "A comparative study of white box, black box and grey box testing techniques," *Int. J. Adv. Comput. Sci. Appl.*, vol. 3, no. 6, 2012.
- 478
- 479 34. Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, Z. Su, and X. Jiao, "Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers."
- 480
- 481 35. J. Liang, Y. Jiang, Y. Chen, M. Wang, C. Zhou, and J. Sun, "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*. ACM, 2018, pp. 809–814.
- 482
- 483

484 © 2019 by the authors. Submitted to *Journal Not Specified* for possible open access  
485 publication under the terms and conditions of the Creative Commons Attribution (CC BY) license  
486 (<http://creativecommons.org/licenses/by/4.0/>).