EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers

Yuanliang Chen, Yu Jiang* {jiangyu198964@126.com},
Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou and hZuo Su
School of Software, Tsinghua University, KLISS, Beijing, China

Xun Jiao {canbyjiaoxun@163.com}
ECE department of Villanova University

Abstract

Fuzzing is widely used for vulnerability detection. There are various kinds of fuzzers with different fuzzing strategies, and most of them perform well on their targets. However, in industrial practice, it is found that the performance of those well-designed fuzzing strategies is challenged by the complexity and diversity of real-world applications. In this paper, we systematically study an ensemble fuzzing approach. First, we define the diversity of base fuzzers in three heuristics: diversity of coverage information granularity, diversity of input generation strategy and diversity of seed selection and mutation strategy. Based on those heuristics, we choose several of the most recent base fuzzers that are as diverse as possible, and propose a globally asynchronous and locally synchronous (GALS) based seed synchronization mechanism to seamlessly ensemble those base fuzzers and obtain better performance. For evaluation, we implement `EnFuzz` based on several widely used fuzzers such as QSYM and FairFuzz, and then test them on LAVA-M and Google's fuzzing-test-suite, which consists of 24 widely used real-world applications. This experiment indicates that, under the same constraints for resources, these base fuzzers perform differently on different applications, while EnFuzz always outperforms others in terms of path coverage, branch coverage and bug discovery. Furthermore, `EnFuzz` found 60 new vulnerabilities in several well-fuzzed projects such as libpng and libjpeg, and 44 new CVEs were assigned.

***Index terms*—** Ensemble Fuzzing, Seed Synchronization

1 Introduction

Fuzzing is one of the most popular software testing techniques for bug and vulnerability detection. There are many fuzzers for academic and industrial usage. The key idea of fuzzing is to generate plenty of inputs to execute the target application and monitor for any anomalies. While each fuzzer develops its own specific fuzzing strategy to generate inputs, there are in general two main types of strategies. One is a generation-based strategy which uses the specification of input format, e.g. grammar, to generate complex inputs. For example, IFuzzer [33] takes a context-free grammar as specification to generate parse trees for code fragments. Radamsa [22] reads sample files of valid data and generates interesting different outputs from them. The other main strategy is a mutation-based strategy. This approach generates new inputs by mutating the existing seeds (good inputs contributing to improving the coverage). Recently, mutation-based fuzzers are proposed to use coverage information of target programs to further improve effectiveness for bug detection. For example, libFuzzer [10] mutates seeds by utilizing the SanitizerCoverage [11] instrumentation to track block coverage, while AFL [39] mutates seeds by using static instrumentation to track edge coverage.

Based on the above mentioned two fuzzers, researchers have performed many optimizations. For example, AFLFast [16] improves the fuzzing strategy of AFL by selecting seeds that exercise low-frequency paths for additional mutations, and FairFuzz [26] optimizes AFL's mutation algorithm to prioritize seeds that hit rare branches. AFLGo [15] assigns more mutation times to the seeds closer to target locations. QSYM [38] uses a practical concolic execution engine to solve complex branches of AFL. All of these optimized fuzzers outperform AFL on their target applications and have already detected a large number of software bugs and security vulnerabilities.

However, when we apply these optimized fuzzers to some real-world applications, these fuzzing strategies are inconsistent in their performance, their effectiveness on different applications varies accordingly. For example, in our evaluation on 24 real-world applications, AFLFast and FairFuzz perform better than AFL on 19 applications, while AFL performs better on the other 5 applications. Compared with AFL, libFuzzer performs better on 17 applications but worse on the other 7 applications. For the parallel mode of fuzzing which is widely-used in industry, AFLFast and FairFuzz only detected 73.5% and 88.2% of the unique bugs of AFL. These results show that the performance of existing fuzzers is challenged by the complexity and diversity of real-world applications. For a given real-world application, we cannot evaluate which fuzzer is better unless we spend significant time analyzing them or running each of these fuzzers one by one. This would waste a lot of human and computing resources [25]. This indicates that many of the current fuzzing strategies have a lack of robustness — the property of being strong and stable consistently in constitution. For industrial prac-

tice, more robust fuzzing strategies are desired when applied across different applications.

In this paper, we systematically study the performance of an ensemble fuzzing approach. First, we define the diversity of base fuzzers focusing on three heuristics: diversity of coverage information granularity, diversity of input generation strategy, as well as diversity of seed mutation and selection strategy. Then, we implement an ensemble architecture with a global asynchronous and local synchronous(GALS) based seed synchronization mechanism to integrate these base fuzzers effectively. To enhance cooperation among different base fuzzers, the mechanism synchronizes interesting seeds(i.e., test cases covering new paths or triggering new crashes) periodically to all fuzzers running on the same target application. At the same time, it maintains a global coverage map to help collect those interesting seeds asynchronously from each base fuzzer.

For evaluation, we implement a prototype of `EnFuzz`, based on several high-performance base fuzzers, including AFL, AFLFast, FairFuzz, QSYM, libFuzzer and Radamsa. All fuzzers are repeatedly tested on two widely used benchmarks — LAVA-M and Google's fuzzer-test-suite, following the kernel rules of evaluating fuzzing guideline[25]. The average number of paths executed, branches covered and unique crashes discovered are used as metrics. The results demonstrate that, with the same resource usage, the base fuzzers perform differently on different applications, while `EnFuzz` consistently and effectively improves the fuzzing performance. For example, there are many cases where the original AFL performs better on some real-world applications than the two optimized fuzzers FairFuzz and AFLFast. In all cases, the ensemble fuzzing always outperforms all other base fuzzers.

Specifically, on Google's fuzzer-test-suite consisting of real-world applications with a code base of 80K-220K LOCs, compared with AFL, AFLFast, FairFuzz, QSYM, libFuzzer and Radamsa, `EnFuzz` discovers 76.4%, 140%, 100%, 81.8%, 66.7% and 93.5% more unique bugs, executes 42.4%, 61.2%, 45.8%, 66.4%, 29.5% and 44.2% more paths and covers 15.5%, 17.8%, 12.9%, 26.1%, 19.9% and 14.8% more branches respectively. For the result on LAVA-M consisting of applications with a code base of 2K-4K LOCs, it outperforms each base fuzzer as well. For further evaluation on more widely used and several well-fuzzed open-source projects such as Libpng and jpeg, `EnFuzz` finds 60 new real vulnerabilities, 44 of which are security-critical vulnerabilities and registered as new CVEs. However, other base fuzzers only detect 35 new vulnerabilities at most.

This paper makes the following main contributions:

1. While many earlier works have mentioned the possibility of using ensemble fuzzing, we are among the first to systematically investigate the practical ensemble fuzzing strategies and the effectiveness of ensemble fuzzing of various fuzzers. We evaluate the performance of typical fuzzers through a detailed empirical study. We define the diversity of base fuzzers and study the effects of diversity on their performance.

2. We implement a concrete ensemble approach with seed synchronization to improve the performance of existing fuzzers. `EnFuzz` shows a more robust fuzzing practice across diverse real world applications. The prototype[1] is also scalable and open-source so as to integrate other fuzzers.

3. We apply `EnFuzz` to fuzz several well-fuzzed projects such as libpng and libjpeg from GitHub, and several commercial products such as `libiec61850` from Cisco. Within 24 hours, 60 new security vulnerabilities were found and 44 new CVEs were assigned, while other base fuzzers only detected 35 new vulnerabilities at most. `EnFuzz` has already been deployed in industrial practice, and more new CVEs are being reported[1].

The rest of this paper is organized as follows: Section 2 introduces related work. Section 3 illustrates ensemble fuzzing by a simple example. Section 4 elaborates ensemble fuzzing, including the base fuzzer selection and ensemble architecture design. Section 5 presents the implementation and evaluation of `EnFuzz`. Section 6 discusses the potential threats of `EnFuzz`, and we conclude in section 7. The appendix shows some empirical evaluations and observations.

2 Related Work

Here below, we introduce the work related to generation-based fuzzing, mutation-based fuzzing, fuzzing in practice and the main differences between these projects. After that we summarize the inspirations and introduce our work.

2.1 Generation-based Fuzzing

Generation-based fuzzing generates a massive number of test cases according to the specification of input format, e.g. a grammar. To fuzz the target applications that require inputs in complex format, the specifications used are crucial. There are many types of specifications. Input model and context-free grammar are the two most common types. Model-based fuzzers [20, 34, 1] follow a model of protocol. Hence, they are able to find more complex bugs by creating complex interactions with the target applications. Peach [20] is one of the most popular model-based fuzzers with both generation and mutation abilities. It develops two key models: the data model determines the format of complex inputs and the state model describes the concrete method for cooperating with fuzzing targets. By integrating fuzzing with models of data and state, Peach works effectively. Skyfire [34] first learns a context-sensitive grammar model, and then it generates massive inputs based on this model.

Some other popular fuzzers [21, 37, 31, 33, 24] generate inputs based on context free grammar. P Godefroid [21] enhances the whitebox fuzzing of complex structured-input applications by using symbolic execution, which directly generates grammar-based constraints whose satisfiability is examined using a custom grammar-based constraint solver. Csmith [37] is designed for fuzzing C-compilers. It generates plenty of random C programs in the C99 standard as the

[1] `https://github.com/enfuzz/enfuzz`

inputs. This tool can be used to generate C programs exploring a typical combination of C-language features while being free of undefined and unspecified behaviors. LAVA [31] generates effective test suites for the Java virtual machine by specifying production grammars. IFuzzer [33] first constructs parse trees based on a language's context-free grammar, then it generates new code fragments according to these parse trees. Radamsa [22] is a widely used generation-based fuzzer. It works by reading sample files of valid data and generating interestingly different outputs from them. Radamsa is an extreme "black-box" fuzzer, it needs no information about the program nor the format of the data. One can pair it with coverage analysis during testing to improve the quality of the sample set during a continuous fuzzing test.

2.2 Mutation-based Fuzzing

Mutation-based fuzzers [23, 17, 2] mutate existing test cases to generate new test cases without any input grammar or input model specification. Traditional mutation-based fuzzers such as zzuf [23] mutate the test cases by flipping random bits with a predefined ratio. In contrast, the mutation ratio of SYMFUZZ [17] is assigned dynamically. To detect bit dependencies of the input, it leverages white-box symbolic analysis on an execution trace, then it dynamically computes an optimal mutation ratio according to these dependencies. Furthermore, BFF [2] integrates machine learning with evolutionary computation techniques to reassign the mutation ratio dynamically.

Other popular AFL family tools [39, 16, 15, 26] apply various strategies to boost the fuzzing process. AFLFast [16] regards the process of target application as a Markov chain. A path-frequency based power schedule is responsible for computing the times of random mutation for each seed. As with AFLFast, AFLGo [15] also proposes a simulated annealing-based power schedule, which helps fuzz the target code. FairFuzz [26] mainly focuses on the mutation algorithm. It only mutates seeds that hit rare branches and it strives to ensure that the mutant seeds hit the rarest one. (Wen Xu et.al.)[36] propose several new primitives , speeding up AFL by 6.1 to 28.9 times. Unlike AFL family tools which track the hit count of each edge, libFuzzer [10] and honggfuzz [5] utilize the SanitizerCoverage instrumentation method provided by the Clang compiler. To track block coverage, they track the hit count of each block as a guide to mutate the seeds during fuzzing. SlowFuzz [30] prioritizes seeds that use more computer resources (e.g., CPU, memory and energy), increasing the probability of triggering algorithmic complexity vulnerabilities. Furthermore, some fuzzers use concolic executors for hybrid fuzzing. Both Driller [32] and QSYM use mutation-based fuzzers to avoid path exploration of symbolic execution, while concolic execution is selectively used to drive execution across the paths that are guarded by narrow-ranged constraints.

2.3 Cluster and Parallel Fuzzing in Industry

Fuzzing has become a popular vulnerability discovery solution in industry [28] and has already found a large number of dangerous bugs and security vulnerabilities across a wide range of systems so far. For example, Google's OSS-Fuzz [4] platform has found more than 1000 bugs in 5 months with thousands of virtual machines [9]. ClusterFuzz is the distributed fuzzing infrastructure behind OSS-Fuzz, and automatically executes libFuzzer powered fuzzer tests on scale [12, 13]. Initially built for fuzzing Chrome at scale, ClusterFuzz integrates multiple distributed libFuzzer processes, and performs effectively with corpus synchronization. ClusterFuzz mainly runs multiple identical instances of libFuzzer on distributed system for one target application. There is no diversity between these fuzzing instances.

In industrial practice, many existing fuzzers also provide a parallel mode, and they work well with some synchronization mechisms. For example, each instance of AFL in parallel mode will periodically re-scan the top-level sync directory for any test cases found by other fuzzers[7, 3]. libFuzzer in parallel will also use multiple fuzzing engines to exchange the corpora[6]. These parallel mode can effectively improve the performance of fuzzer. In fact, the parallel mode can be seen as a special example of ensemble fuzzing which uses multiple same base fuzzers. However, all these base fuzzers have a lack of diversity when using the same fuzzing strategy.

2.3.1 Main Differences

Unlike the previous works, we are not proposing a new concrete generation-based or mutation-based fuzzing strategy. Nor do we run multiple identical fuzzers with multiple cores or machines. Instead, inspired by the seed synchronization of ClusterFuzz and AFL in parallel mode, we systematically study the possibility of the ensemble fuzzing of diverse fuzzers mentioned in the earlier works. Referred to the kernel descriptions of the evaluating fuzzing guidelines[25], we empirically evaluate most state-of-the-art fuzzers, and identify some valuable results, especially for their performance variation across different real applications. To generate a stronger ensemble fuzzer, we choose multiple base fuzzers that are as diverse as possible based on three heuristics. We then implement an ensemble approach with global asynchronous and local synchronous based seed synchronization.

3 Motivating Example

To investigate the effectiveness of ensemble fuzzing, we use a simple example in Figure 1 which takes two strings as input, and crashes when one of the two strings is "Magic Str" and the other string is "Magic Num".

Many existing fuzzing strategies tend to be designed with certain preferences. Suppose that we have two different fuzzers $fuzzer_1$ and $fuzzer_2$: $fuzzer_1$ is good at solving the "Magic Str" problem, so it is better for reaching targets T1 and T3, but fails to reach targets T2 and T4. $fuzzer_2$ is good at solving the "Magic Num" problem so it is better for reaching targets T2 and T6, but fails to reach targets T1 and T5. If we use these two fuzzers separately, we can only cover one path and two branches. At the same time, if we use them simultaneously and ensemble their final fuzzing results without seed synchronization, we can cover two paths and four

```
void crash(char* A, char* B){
    if (A == "Magic Str"){              => T1
        if (B == "Magic Num") {
            bug();                      => T4
        }else{
            normal();                   => T3
        }
    }else if (A == "Magic Num"){        => T2
        if (B == "Magic Str"){
            bug();                      => T5
        }else{
            normal();                   => T6
        }
    }
}
```

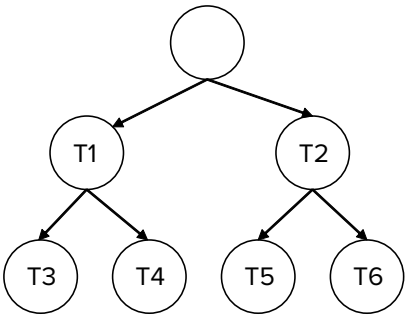


Figure 1: Motivating example of ensemble fuzzing with seed synchronization.

branches. However, if we ensemble these two fuzzers with some synchronization mechanisms throughout the fuzzing process, then, once $fuzzer_1$ reaches T1, it synchronizes the seed that can cover T1 to $fuzzer_2$. As a result, then, with the help of this synchronized seed, $fuzzer_2$ can also reach T1, and because of its ability to solve the "Magic Num" problem, $fuzzer_2$ can further reach T4. Similarly, with the help of the seed input synchronized by $fuzzer_2$, $fuzzer_1$ can also further reach T2 and T5. Accordingly, all four paths and all six branches can be covered through this ensemble approach.

Table 1: covered paths of each fuzzing option

Tool	T1-T3	T1-T4	T2-T5	T2-T6
fuzzer1	✓			
fuzzer2				✓
ensemble fuzzer1 and fuzzer2 without seed synchronization	✓			✓
ensemble fuzzer1 and fuzzer2 with seed synchronization	✓	✓	✓	✓

The ensemble approach in this motivating example works based on the following two hypotheses: (1) $fuzzer_1$ and $fuzzer_2$ expert in different domains; (2) the interesting seeds can be synchronized to all base fuzzers in a timely way. To satisfy the above hypotheses as much as possible, successful ensemble fuzzers rely on two key points: (1) the first is to select base fuzzers with great diversity (as yet to be well-defined); (2) the second is a concrete synchronization mechanism to enhance effective cooperation among those base fuzzers.

4 Ensemble Fuzzing

For an ensemble fuzzing, we need to construct a set of base fuzzers and seamlessly combine them to test the same target application together. The overview of this approach is presented in Figure 2. When a target application is prepared for fuzzing, we first choose several existing fuzzers as base fuzzers. The existing fuzzing strategies of any single fuzzer are usually designed with preferences. In real practice, these preferences vary greatly across different applications. They can be helpful in some applications, but may be less effective on other applications. Therefore, choosing base fuzzers with more diversity can lead to better ensemble performance. After the base fuzzer selection, we integrate fuzzers with the globally asynchronous and locally synchronous based seed synchronization mechanism so as to monitor the fuzzing status of these base fuzzers and share interesting seeds among them. Finally, we collect crash and coverage information and feed this information into the fuzzing report.

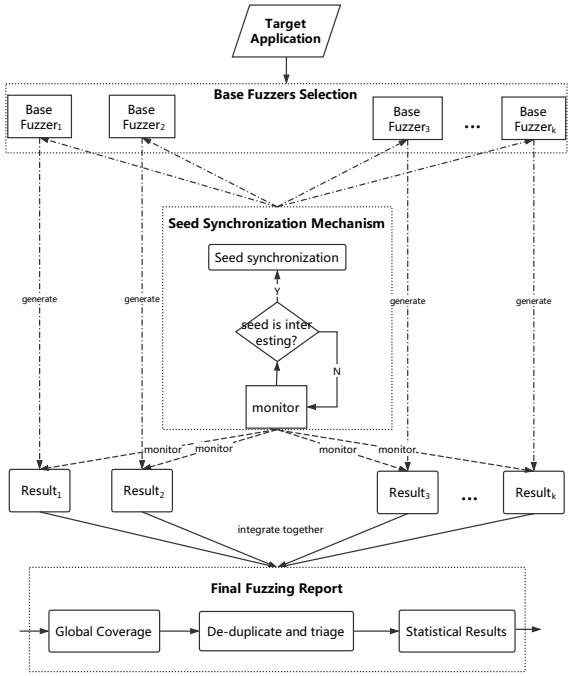


Figure 2: The overview of ensemble fuzzing consists of base fuzzer selection and ensemble architecture design. The base fuzzer selection contains the diversity heuristic definition, and the architecture design includes the seed synchronization mechanism as well as final fuzzing report.

4.1 Base Fuzzer Selection

The first step in ensemble fuzzing is to select a set of base fuzzers. These fuzzers can be generation-based fuzzers, e.g. Peach and Radamsa, or mutation-based fuzzers, e.g. libFuzzer and AFL. We can randomly choose some base fuzzers, but selecting base fuzzers with well-defined diversity improves the performance of an ensemble fuzzer.

We classify the diversity of base fuzzers according to three heuristics: seed mutation and selection strategy diversity, coverage information granularity diversity, inputs generation strategy diversity. The diversity heuristics are as follows:

1. Seed mutation and selection strategy based heuristic: the diversity of base fuzzers can be determined by the variability of seed mutation strategies and seed selection strategies. For example, AFLFast selects seeds that exercise low-frequency paths and mutates them more times, FairFuzz strives to ensure that the mutant seeds hit the rarest branches.

2. Coverage information granularity based heuristic: many base fuzzers determine interesting inputs by tracking different coverage information. Hence, the coverage information is critical, and different kinds of coverage granularity tracked by fuzzers enhances diversity. For example, libFuzzer guides seed mutation by tracking block coverage while AFL tracks edge coverage.

3. Input generation strategy based heuristic: fuzzers with different input generation strategies are suitable for different tasks. For example, generation-based fuzzers use the specification of input format to generate test cases, while the mutation-based fuzzers mutate initial seeds by tracking code coverage. So the generation-based fuzzers such as Radamsa perform better on complex format inputs and the mutation-based fuzzers such as AFL prefer complex logic processing.

Based on these three basic heuristics, we should be able to select a diverse set of base fuzzers with large diversity. It is our intuition that the diversity between the fuzzers following in two different heuristics is usually larger than the fuzzers that follows in the same heuristic. So, the diversity among the AFL family tools should be the least, while the diversity between Radamsa and AFL, between Libfuzzer and AFL, and between QSYM and AFL is should be greater. In this paper, we select base fuzzers manually based on the above heuristics. the base fuzzers will be dynamically selected according to the real-time coverage information.

4.2 Ensemble Architecture Design

After choosing base fuzzers, we need to implement a suitable architecture to integrate them together. As presented in Figure 2, inspired by the seed synchronization of AFL in parallel mode, one core mechanism is designed — the globally asynchronous and locally synchronous (GALS) based seed synchronization mechanism. The main idea is to identify the interesting seeds (seeds that can cover new paths or new branches or can detect new unique crashes) from different base fuzzers asynchronously and share those interesting seeds synchronously among all fuzzing processes.

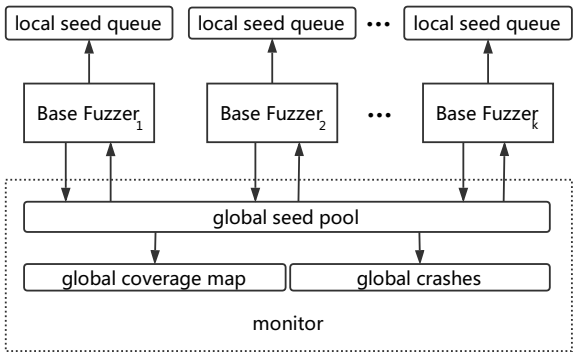


Figure 3: The data structure of global asynchronous and local synchronous based seed synchronization mechanism.

ALGORITHM 1: Action of local base fuzzer

```
   Input : Local seed pool of base fuzzer queue
1  repeat
2      foreach seed s of the queue do
3          s' = Mutate(s);
4          Cover = Run(s');
5          // if seeds s' causes new crash or have new
             // coverage, then store it in own seed pool and
             // push it to the global seed pool asynchronously;
6          if Cover.causeCrash() then
7              crashes.push(s');
8              queue.push(s');
9              GlobalSeedPool.push(s');
10         else if Cover.haveNewCoverage() then
11             queue.push(s');
12             GlobalSeedPool.push(s');
13         end
14     end
15 until timeout or abort-signal;
   Output: Global crashing seeds crashes
```

This seed synchronization mechanism employs a global-local style data structure as shown in Figure 3. The local seed queue is maintained by each base fuzzer, while the global pool is maintained by the monitor for sharing interesting seeds among all base fuzzers. In ensemble fuzzing, the union of these base fuzzers' results is needed to identify interesting seeds during the whole fuzzing process. Accordingly, the global coverage map is designed, and any new paths or new branches covered by the interesting seeds will be added into this global map. This global map can not only help decide which seeds to be synchronized, but also help deduplicate and triage the results. Furthermore, to output the final fuzzing report after completing all fuzzing jobs, any interesting seeds which contribute to triggering unique crashes will be stored in the global crashes list.

First, let us take a look at the seed synchronization solution

of the base fuzzer, which mainly describes how base fuzzers contribute the interesting seeds asynchronously to the global pool. As presented in lines 2-4 of algorithm 1, for each single base fuzzer, it works with a local input seed queue and runs a traditional continuous fuzzing loop. It has three main steps: (1) Select input seeds from the queue, (2) mutate the selected input seeds to generate new candidate seeds, (3) run the target program with the candidate seeds, track the coverage and report vulnerabilities. Once the candidate seeds have new coverage or cause unique crashes, they will be regarded as interesting seeds and be pushed asynchronously into the global seed pool, as presented in lines 6-12.

ALGORITHM 2: Action of global monitor *sync*

```
   Input : Base fuzzers list BaseFuzzers[]
           Initial seeds S
           Synchronization period period
1  // set up each base fuzzers ;
2  foreach base fuzzer f of the BaseFuzzers[] do
3  |   fuzzer.setup();
4  end
5  // set up thread monitor for monitoring ;
6  monintor.setup();
7  GlobalCover.initial();
8  GlobalSeedPool.initial();
9  GlobalSeedPool.push(S);
10 repeat
11 |   foreach seed s of the GlobalSeedPool do
12 |   |   // Skip synchronized seeds ;
13 |   |   if s.isSync() == False then
14 |   |   |   foreach base fuzzer f of the BaseFuzzers[] do
15 |   |   |   |   Cover = f.run(s) ;
16 |   |   |   |   // update the global coverage ;
17 |   |   |   |   newCover =
   |   |   |   |    (Cover ∪ GlobalCover) − GlobalCover ;
18 |   |   |   |   GlobalCover = Cover ∪ GlobalCover;
19 |   |   |   |   // synchronize the seed s to base fuzzer f ;
20 |   |   |   |   if Cover.causeCrash() and
   |   |   |   |    !newCover.isEmpty() then
21 |   |   |   |   |   crashes.push(s);
22 |   |   |   |   |   f.queue.push(s);
23 |   |   |   |   else if !newCover.isEmpty() then
24 |   |   |   |   |   f.queue.push(s);
25 |   |   |   |   else
26 |   |   |   |   |   continue;
27 |   |   |   |   end
28 |   |   |   end
29 |   |   else
30 |   |   |   continue;
31 |   |   end
32 |   |   s.setSync(True);
33 |   end
34 |   // waiting until next seed synchronization ;
35 |   sleep(period);
36 until timeout or abort-signal;
   Output: Crashing seeds crashes
```

Second, let us see the seed synchronization solution of the monitor process, which mainly describes how the monitor process synchronously dispatches the interesting seeds in the global pool to the local queue of each base fuzzer. When all base fuzzers are established, a thread named `monitor` will be created for monitoring the execution status of these fuzzing jobs, as in lines 2-6 of algorithm 2. It initializes the global coverage information to record the global fuzzing status of target applications by all the base fuzzer instances and then creates the global seed pool with the initial seeds, as in lines 7-9 of algorithm 2. It then runs a continuous periodically synchronizing loop — each base fuzzer will be synchronously dispatched with the interesting seeds from the global seed pool. Each base fuzzer will incorporate the seeds into its own local seed queue, once the seeds are deemed to be interesting seeds (seeds contribute to the coverage or crash and has not been generated by the local fuzzer), as in line 15-24 . To lower the overhead of seed synchronization, a thread `monitor` is designed to work periodically. Due to this globally asynchronous and locally synchronous based seed synchronization mechanism, base fuzzers cooperate effectively with each other as in the motivating example in Figure 1.

5 Evaluation

To present the effectiveness of ensemble fuzzing, we first implement several prototypes of ensemble fuzzer based on the state-of-the-art fuzzers. Then, we refer to some kernel descriptions of evaluating fuzzing guideline [25]. We conduct thorough evaluations repeatedly on LAVA-M and Google's fuzzer-test-suite, several well-fuzzed open-source projects from GitHub, and several commercial products from companies. Finally, according to the results, we answer the following three questions: (1) Can ensemble fuzzer perform better? (2) How do different base fuzzers affect Enfuzz? (3) How does Enfuzz perform on real-world applications

5.1 Ensemble Fuzzer Implementation

We implement ensemble fuzzing based on six state-of-the-art fuzzers, including three edge-coverage guided mutation-based fuzzers – AFL, AFLFast and FairFuzz, one block-coverage guided mutation-based fuzzer – libFuzzer, one generation-based fuzzer – Radamsa and one most recently hybrid fuzzer – QSYM. These are chosen as the base fuzzers for the following reasons (Note that `EnFuzz` is not limited to these six and other fuzzers can also be easily integrated, such as honggfuzz, ClusterFuzzer etc.):

- Easy integration. All the fuzzers are open-source and have their core algorithms implemented precisely. It is easy to integrate those existing fuzzers into our ensemble architecture. We do not have to implement them on our own, which eliminates any implementation errors or deviations that might be introduced by us.

- Fair comparison. All the fuzzers perform very well and are the latest and widely used fuzzers, as is seen by their comparisons with each other in prior literature, for example, QSYM outperforms similar fuzzers such as Angora[18] and VUzzer. We can evaluate their performance on real-world applications without modification.

- Diversity demonstration. All these fuzzers have different fuzzing strategies and reflect the diversity among

Table 2: Diversity among these base fuzzers

Tool	diversity compared with AFL
AFLFast	Seed mutation and selection strategy based rule: the times of random mutation for each seed is computed by a Markov chain model. The seed selection strategy is different.
FairFuzz	Seed mutation and selection strategy based rule: only mutates seeds which hit rare branches and strives to ensure the mutant seeds hit the rarest one. The seed mutation strategy is different.
libFuzzer	Coverage information granularity based rule: libFuzzer mutates seeds by utilizing the SanitizerCoverage instrumentation, which supports tracking block coverage; while AFL uses static instrumentation with a bitmap to track edge coverage. The coverage information granularity is different.
Radamsa	Input generation strategy based rule: Radamsa is a widely used generation-based fuzzer which generates different inputs sample files of valid data. The input generation strategy is different.
QSYM	QSYM is a practical fast concolic execution engine tailored for hybrid fuzzing. It makes hybrid fuzzing scalable enough to test complex, real-world applications.

correspondence with the three base diversity heuristics mentioned in section 4.1: coverage information granularity diversity, input generation strategy diversity, seed mutation and selection strategy diversity. The concrete diversity among these base fuzzers is listed in Table 2.

To demonstrate the performance of ensemble fuzzing and the influence of diversity among base fuzzers, five prototypes are developed. (1) `EnFuzz-A`, an ensemble fuzzer only based on AFL, AFLFast and FairFuzz. (2) `EnFuzz-Q`, an ensemble fuzzer based on AFL, AFLFast, FairFuzz and QSYM, a practical concolic execution engine is included. (3) `EnFuzz-L`, an ensemble fuzzer based on AFL, AFLFast, FairFuzz and libFuzzer, a block-coverage guided fuzzer is included. (4) `EnFuzz`, an ensemble fuzzer based on AFL, AFLFast, libFuzzer and Radamsa, a generation-based fuzzer is further added .(5) `EnFuzz`$^-$, with the ensemble of same base fuzzers (AFL, AFLFast and FairFuzz), but without the seed synchronization, to demonstrate the effectiveness of the global asynchronous and local synchronous based seed synchronization mechanism. During implementation of the proposed ensemble mechanism, we address the following challenges:

1) Standard Interface Encapsulating The interfaces of these fuzzers are different. For example, AFL family tools use the function `main`, but libFuzzer use a function `LLVMFuzzerTestOneInput`. Therefore, it is hard to ensemble them together. We design a standard interface to encapsulate the complexity of different fuzzing tools. This standard interface takes seeds from the file system, and writes the results back to the file system. All base fuzzers receive inputs and produce results through this standard interface, through which different base fuzzers can be ensembled easily.

2) libFuzzer Continuously Fuzzing The fuzzing engine of libFuzzer will be shut down when it finds a crash, while other tools continue fuzzing until manually closed. It is unfair to compare libFuzzer with other tools when the fuzzing time is different. The persistent mode of AFL is a good solution to this problem. Once AFL sets up, the fuzzer parent will `fork` and `execve` a new process to fuzz the target. When the target process crashes, the parent will collect the crash and resume the target, then the process simply loops back to the start. Inspired by the AFL persistent mode, we set up a thread named `Parent` to monitor the state of libFuzzer. Once it shuts down, `Parent` will resume the libFuzzer.

3) Bugs De-duplicating and Triaging We develop a tool for crash analysis. We compile all the target applications with AddressSanitizer, and test them with the crash samples. When the target applications crash, the coredump file, which consists of the recorded state of the working memory will be saved. Our tool first loads coredump files, then gathers the frames of each crash; finally, it identifies two crashes as identical if and only if the top frame is identical to the other frame. The method above is prone to underestimating bugs. For example, two occurrences of heap overflow may crash at the cleanup function at exit. However, the target program is instrumented with AddressSanitizer. As the program terminates immediately when memory safety problems occur, the top frame is always relevant to the real bug. In practice, the original duplicate unique crashes have been drastically deduplicated to a humanly check-able number of unique bugs, usually without duplication. Even though there are some extreme cases that different top frames for one bug, the result can be further refined by manual crash analysis.

4) Seeds effectively Synchronizing The implementation of the seed synchronization mechanism: all base fuzzers have implemented the communication logic following the standard interface. Each base fuzzer will put interesting seeds into its own local seed pool, and the monitor thread `sync` will periodically make each single base fuzzer pull synchronized seeds from the global seed pool through a communication channel. This communication channel is implemented based on file system. A shorter period consumes too many resources, which leads to a decrease in fuzzing performance. A longer period will make seed synchronizing untimely, which also affects the performance. After multiple attempts with different values, it is found that the synchronization interval affects the performance at the beginning of fuzzing, while little impact was observed in the long term. The interval of 120s is identified with the fastest convergence.

5.2 Data and Environment Setup

Firstly, we evaluate ensemble fuzzing on LAVA-M [19], which consistis of four buggy programs, file, base64, md5sum and who. LAVA-M is a test suite that injects hard-to-find bugs in Linux utilities to evaluate bug-finding techniques. Thus the test is adequate for demonstrating the effectiveness of ensemble fuzzing. Furthermore, to reveal the practical performance of ensemble fuzzing, we also evaluate

our work based on fuzzer-test-suite [8], a widely used benchmark from Google. The test suite consists of popular open-source real-world applications. This benchmark is chosen to avoid the potential bias of the cases presented in literature, and for its great diversity, which helps demonstrate the performance variation of existing base fuzzers.

We refer to the kernel criteria and settings of evaluation from the fuzzing guidelines [25], and integrate the three widely used metrics from previous literature studies to compare the results on these real-world applications more fairly, including the number of paths, branches and unique bugs. To get unique bugs, we use crash's stack backtraces to deduplicate unique crashes, as mentioned in the previous subsection. The initial seeds for all experiments are the same. We use the test cases originally included in their applications or empty seed if such initial seeds do not exist.

The experiment on fuzzer-test-suite is conducted ten times in a 64-bit machine with 36 cores (Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz), 128GB of main memory, and Ubuntu 16.04 as the host OS with SMT enabled. Each binary is hardened by AddressSanitizer [11] to detect latent bugs. First, we run each base fuzzer for 24 hours with one CPU core in single mode. Next, since `EnFuzz-L`, `EnFuzz` and `EnFuzz-Q` need at least four CPU cores to ensemble these four base fuzzers, we also run each base fuzzer in parallel mode for 24 hours with four CPU cores. In particular, `EnFuzz-A` and `EnFuzz`$^-$ only ensembles three types of base fuzzers (AFL, AFLFast and FairFuzz). To use the same resources, we set up two AFL instances, one AFLFast instance and one FairFuzz instance. This experimental setup ensures that the computing resources usage of each ensemble fuzzer is the same as any base fuzzers running in parallel mode. Due to the large amount of data and the page limitation, we include the variation of those statistical test results in our GitHub. In fact. most metrics converged to similar values during multithreaded fuzzing. The variation of those statistical test results is small (between -5% 5%), so we just use the averages in this paper.

5.3 Preliminary Evaluation on LAVA-M

We first evaluate ensemble fuzzing on LAVA-M, which has been used for testing other systems such as Angora, T-Fuzz and QSYM, and QSYM shows the best performance. We run `EnFuzz-Q` (which ensembles AFL, AFLFast, FairFuzz and QSYM) on the LAVA-M dataset. To demonstrate its effectiveness, we also run each base fuzzer using the same resources — four instances of AFL in parallel mode, four instances of AFLFast in parallel mode, four instances of FairFuzz in parallel mode, QSYM with four CPU cores used in parallel mode (two instances of concolic execution engine and two instances of AFL). To identify unique bugs, we used built-in bug identifiers provided by the LAVA project. The results are presented in Table 3, 4 and 5, which show the number of paths executed, branches covered and unique bugs detected by AFL, AFLFast, FairFuzz, QSYM, `EnFuzz-Q`.

From Tables 3, 4 and 5, we found that AFL, AFLFast and FairFuzz perform worse due to the complexity of their branches. The practical concolic execution engine helps QSYM solve complex branches and find significantly more bugs. The base code of the four applications in LAVA-M are small (2K-4K LOCs) and concolic execution could work well on them. However, real projects have code bases that easily reach 10k LOCs. Concolic execution might perform worse or even get hanged, as presented in the latter subsections. Furthermore, when we ensemble AFL, AFLFast, FairFuzz and QSYM together with the GALS based seed synchronization mechanism – `EnFuzz-Q` always performs the best in both coverage and bug detection. In total, compared with AFL, AFLFast, FairFuzz and QYSM, `EnFuzz-Q` executes 44%, 45%, 43% and 7.7% more paths, covers 195%, 215%, 194% and 5.8% more branches, and detectes 8314%, 19533%, 12989% and 0.68% more unique bugs respectively. From these preliminary statistics, we can determine that the performance of fuzzers can be improved by our ensemble approach.

Table 3: Number of paths covered by AFL, AFLFast, FairFuzz, QSYM and `EnFuzz-Q` on LAVA-M.

Project	AFL	AFLFast	FairFuzz	QSYM	`EnFuzz-Q`
base64	1078	1065	1080	1643	**1794**
md5sum	589	589	601	1062	**1198**
who	4599	4585	4593	5621	**5986**
uniq	476	453	471	693	**731**
total	6742	6692	6745	9019	**9709**

Table 4: Number of branches covered by AFL, AFLFast, FairFuzz, QSYM and `EnFuzz-Q` on LAVA-M.

Project	AFL	AFLFast	FairFuzz	QSYM	`EnFuzz-Q`
base64	388	358	389	960	**993**
md5sum	230	208	241	2591	**2786**
who	813	791	811	1776	**1869**
uniq	1085	992	1079	1673	**1761**
total	2516	2349	2520	7000	**7409**

Table 5: Number of bugs found by AFL, AFLFast, FairFuzz, QSYM and `EnFuzz-Q` on LAVA-M.

Project	AFL	AFLFast	FairFuzz	QSYM	`EnFuzz-Q`
base64	1	1	0	41	**42**
md5sum	0	0	1	**57**	**57**
who	2	0	1	1047	**1053**
uniq	11	5	7	25	**26**
total	14	6	9	1170	**1178**

5.4 Evaluation on Google's fuzzer-test-suite

While LAVA-M is widely used, Google's fuzzer-test-suite is more practical with many more code lines and containing real-world bugs. To reveal the effectiveness of ensemble fuzzing, we run `EnFuzz` (which only ensembles AFL, AFLFast, LibFuzzer and Radamsa) on all of the 24 real-world applications of Google's fuzzer-test-suite for 24 hours

10 times. As a comparison, we also run each base fuzzer in parallel mode with four CPU cores used. To identify unique bugs, we used stack backtraces to deduplicate crashes. The results are presented in Tables 6, 7 and 8, which shows the average number of paths executed, branches covered and unique bugs detected by AFL, AFLFast, FairFuzz, LibFuzzer, Radamsa, QSYM and EnFuzz respectively.

Table 6: Average number of paths covered by each tool on Google's fuzzer-test-suite for ten times.

Project	AFL	AFLFast	FairFuzz	LibFuzzer	Radamsa	QSYM	EnFuzz
boringssl	3286	2816	3393	5525	3430	2973	**7136**
c-ares	146	116	146	191	146	132	**253**
guetzli	3248	2550	1818	3844	3342	2981	**4508**
lcms	1682	1393	1491	1121	1416	1552	**2433**
libarchive	12842	10111	12594	22597	12953	11984	**31778**
libssh	110	102	110	362	110	149	**377**
libxml2	14888	13804	14498	28797	17360	13172	**35983**
openssl-1.0.1	3992	3501	3914	2298	3719	3880	**4552**
openssl-1.0.2	4090	3425	3956	2304	3328	3243	**4991**
openssl-1.1.0	4051	3992	4052	2638	3593	4012	**4801**
pcre2	79581	66894	71671	59616	78347	60348	**85386**
proj4	342	302	322	509	341	323	**709**
re2	12093	10863	12085	15682	12182	10492	**17155**
woff2	23	16	20	447	22	24	**1324**
freetype2	19086	18401	20655	25621	18609	17707	**27812**
harfbuzz	12398	11141	14381	16771	11021	12557	**16894**
json	1096	963	721	1081	1206	1184	**1298**
libjpeg	1805	1579	2482	1486	1632	1636	**2638**
libpng	582	568	587	586	547	606	**781**
llvm	8302	8640	9509	10169	8019	7040	**10935**
openthread	268	213	230	1429	266	365	**1506**
sqlite	298	322	294	580	413	300	**636**
vorbis	1484	1548	1593	1039	1381	1496	**1699**
wpantund	4914	5112	5691	4881	4891	4941	**5823**
Total	190607	168372	186213	209574	188274	163097	**271408**
Improvement	–	11% ↓	2% ↓	9% ↑	1% ↓	14%↓	**42%** ↑

The first six columns of Table 6 reveal the issue of the performance variation in those base fuzzers, as they perform variously on different applications. Comparing AFL family tools, AFL performs better than the other two optimized fuzzers on 14 applications. Compared with AFL, libFuzzer performs better on 15 applications, but worse on 9 applications. Radamsa performs better on 8 applications, but also worse on 16 applications. QSYM performs better on 9 applications, but also worse on 15 applications. Table 7 and Table 8 show similar results on branch coverage and bugs.

From Table 6, it is interesting to see that compared with those optimized fuzzers based on AFL (AFLFast, FairFuzz, Radamsa and QSYM), original AFL performs the best on 14 applications in parallel mode with 4 CPU cores. For the total number of paths executed, AFL performs the best and AFLFast performs the worst in parallel mode. While in single mode with one CPU core used, the situation is exactly the opposite, and the original AFL only performs the best on 5 applications, as presented in Table 14 of the appendix.

The reason for performance degradation of these optimizations in parallel mode is that their studies lack the consideration for synchronizing the additional guiding information. Take AFLFast for example, it models coverage-based fuzzing as Markov Chain, and the times of random mutation for each seed will be computed by a power scheduler. This strategy works well in single mode, but it would fail in parallel mode because the statistics of each fuzzer's scheduler are limited in current thread. Our evaluation demonstrates that many optimized fuzzing strategies could be useful in single mode, but fail in the parallel mode even if this is the mode widely used in industry practice. This experiment has been missing by many prior literature studies. A potential solution for this degradation is to synchronize the additional guiding information in their implementation, similar to the work presented in PAFL[27].

Table 7: Average number of branches covered by each tool on n Google's fuzzer-test-suite for ten times.

Project	AFL	AFLFast	FairFuzz	LibFuzzer	Radamsa	QSYM	EnFuzz
boringssl	3834	3635	3894	3863	3880	3680	**4108**
c-ares	**285**	276	**285**	202	**285**	**285**	**285**
guetzli	3022	2723	1514	**4016**	3177	3011	3644
lcms	3985	3681	3642	3015	2857	3731	**4169**
libarchive	10580	9267	8646	8635	11415	9416	**13949**
libssh	614	614	614	573	614	**636**	614
libxml2	15204	14845	14298	13346	19865	14747	**21899**
openssl-1.0.1	4011	3967	3996	3715	4117	4032	**4673**
openssl-1.0.2	4079	4004	4021	3923	4074	3892	**4216**
openssl-1.1.0	9125	9075	9123	8712	9017	9058	**9827**
pcre2	50558	48004	49430	36539	51881	36208	**53912**
proj4	267	267	267	798	267	261	**907**
re2	17918	17069	17360	16001	17312	16323	**19688**
woff2	120	120	120	2785	120	121	**3945**
freetype2	53339	52404	56653	57325	52715	48547	**58192**
harfbuzz	38163	36313	43077	39712	37959	38194	**44708**
json	7048	6622	5138	6583	7231	7169	**7339**
libjpeg	12345	11350	15688	10342	12009	11468	**17071**
libpng	4135	4393	4110	4003	3961	4085	**4696**
llvm	55003	56619	58306	57021	54312	48008	**62918**
openthread	3109	2959	2989	5421	3102	3634	**5579**
sqlite	2850	2847	2838	3123	3012	2853	**3216**
vorbis	12136	13524	13053	10032	11234	12849	**14318**
wpantund	40667	40867	41404	39816	40317	40556	**43217**
Total	352397	345445	360466	339501	354733	322764	**407090**
Improvement	–	1% ↓	2% ↓	3% ↑	0.6% ↓	8%↓	**16%** ↑

Table 8: Average number of unique bugs found by each tool on n Google's fuzzer-test-suite for ten times.

Project	AFL	AFLFast	FairFuzz	LibFuzzer	Radamsa	QSYM	EnFuzz
boringssl	0	0	0	**1**	0	0	**1**
c-ares	**3**	2	**3**	1	2	2	**3**
guetzli	0	0	0	**1**	0	0	**1**
lcms	1	1	1	**2**	1	1	**2**
libarchive	0	0	0	**1**	0	0	**1**
libssh	0	0	0	1	0	1	**2**
libxml2	1	1	1	**3**	2	1	**3**
openssl-1.0.1	3	2	3	2	2	3	**4**
openssl-1.0.2	5	4	4	1	5	5	**6**
openssl-1.1.0	5	5	5	3	4	5	**6**
pcre2	6	4	5	2	5	4	**8**
proj4	2	0	1	1	1	1	**3**
re2	1	0	1	1	0	1	**2**
woff2	1	0	0	**2**	1	1	1
freetype2	0	0	0	0	0	0	0
harfbuzz	0	0	**1**	**1**	0	0	**1**
json	2	1	0	1	**3**	2	**3**
libjpeg	0	0	0	0	0	0	0
libpng	0	0	0	0	0	0	0
llvm	1	1	**2**	**2**	1	1	**2**
openthread	0	0	0	**4**	0	0	**4**
sqlite	0	0	0	**3**	1	1	**3**
vorbis	3	**4**	3	3	3	**4**	**4**
wpantund	0	0	0	0	0	0	0
Total	34	25	30	37	31	33	**60**
Improvement	–	26% ↓	12% ↑	6% ↓	9% ↑	3%↓	**76%** ↑

From the fifth columns of Table 6 and Table 14, we find that compared with Radamsa in single mode, the improve-

ment achieved by Radamsa is limited in parallel mode. There are two main reasons: (1) Too many useless inputs generated by Radamsa slow down the seed-sharing efficiency among all instances of AFL. This seed-sharing mechanism does not exist in single mode. (2) Some interesting seeds can be created in parallel mode and shared among all instances of AFL. These seeds overlap with the inputs generated by Radamsa. So this improvement is limited in parallel mode.

For the EnFuzz which integrates AFL, AFLFast, LibFuzzer and Radamsa as base fuzzers and, compared with AFL, AFLFast, FairFuzz, QSYM, LibFuzzer and Radamsa, it shows the strongest robustness and always performs the best. In total, it discovers 76.4%, 140%, 100%, 81.8%, 66.7% and 93.5% more unique bugs, executes 42.4%, 61.2%, 45.8%, 66.4%, 29.5% and 44.2% more paths and covers 15.5%, 17.8%, 12.9%, 26.1%, 19.9% and 14.8% more branches respectively. These statistics demonstrate that it helps mitigate performance variation and improves robustness and performance by the ensemble approach with globally asynchronous and locally synchronous seed synchronization mechanism.

5.5 Effects of Different Fuzzing Integration

To study the effects of the globally asynchronous and locally synchronous based seed synchronization mechanism, we conduct a comparative experiment on EnFuzz$^-$and EnFuzz-A, both ensemble the same base fuzzers (2 instances of AFL, 1 instance of AFLFast, 1 instance of FairFuzz) in parallel mode with four CPU cores. To study the effects of different base fuzzers on ensemble fuzzing, we also run EnFuzz-Q, EnFuzz-L and EnFuzz on Google's fuzzer-test-suite for 24 hours 10 times. To identify unique bugs, we used stack backtraces to deduplicate crashes. The results are presented in Tables 9, 10 and 11, which shows the average number of paths executed, branches covered and unique bugs detected by EnFuzz$^-$, EnFuzz-A, EnFuzz-Q, EnFuzz-L, and EnFuzz, respectively.

Compared with EnFuzz-A, EnFuzz$^-$ which ensembles the same base fuzzers AFL, AFLFast and FairFuzz, but does not implement the seed synchronization mechanism. EnFuzz$^-$ performs much worse on all applications. In total, it only executes 68.5% paths, covers 78.3% branches and detects 32.4% unique bugs of EnFuzz-A. These statistics demonstrate that the globally asynchronous and locally synchronous based seed synchronization mechanism is critical to the ensemble fuzzing.

For EnFuzz-A, which ensembles AFL, AFLFast and FairFuzz as base fuzzers and implements the seed synchronization with global coverage map, compared with AFL, AFLFast and FairFuzz running in parallel mode with four CPU cores used (as shown in Table 6, Table 7 and Table 8), it always executes more paths and covers more branches on all applications. In total, it covers 11.3%, 25.9% and 13.9% more paths, achieves 7.2%, 9.3% and 4.8% more covered branches, and triggers 8.8%, 48% and 23% more unique bugs. It reveals that the robustness and performance can be improved even when the diversity of base fuzzers is small.

For the EnFuzz-Q which integrates AFL, AFLFast, FairFuzz and QYSM as base fuzzers, the results are shown in the fourth columns of Tables 9, 10 and 11. Compared with EnFuzz-A, EnFuzz-Q covers 1.1% more paths, executes 1.0% more branches and triggers 10.8% more unique bugs than EnFuzz-A. The improvement is significantly smaller on Google's fuzzer-test-suite than on LAVA-M.

Table 9: Average number of paths covered by each Enfuzz on Google's fuzzer-test-suite for ten times.

Project	EnFuzz$^-$	EnFuzz-A	EnFuzz-Q	EnFuzz-L	EnFuzz
boringssl	2590	4058	3927	6782	**7136**
c-ares	149	167	159	251	**253**
guetzli	2066	3501	3472	4314	**4508**
lcms	1056	1846	1871	2253	**2433**
libarchive	4823	14563	14501	28531	**31778**
libssh	109	140	152	**377**	**377**
libxml2	11412	19928	18738	33940	**35983**
openssl-1.0.1	3496	4015	4095	4417	**4552**
openssl-1.0.2	3949	4976	**5012**	4983	4991
openssl-1.1.0	3850	4291	4383	4733	**4801**
pcre2	57721	81830	82642	84681	**85386**
proj4	362	393	399	708	**709**
re2	9053	13019	14453	17056	**17155**
woff2	19	25	24	1314	**1324**
freetype2	17692	22512	20134	26421	**27812**
harfbuzz	10438	14997	15019	16328	**16894**
json	648	1101	1183	1271	**1298**
libjpeg	1395	2501	2475	2588	**2638**
libpng	480	601	652	706	**781**
llvm	7953	9706	9668	10883	**10935**
openthread	197	281	743	1489	**1506**
sqlite	279	311	325	598	**636**
vorbis	928	1604	1639	1673	**1699**
wpantund	4521	5718	5731	5797	**5823**
Total	145186	212084	211397	262094	**271408**
Improvement	–	46% ↑	48% ↑	80% ↑	**87%** ↑

Table 10: Average number of branches covered by each Enfuzz on Google's fuzzer-test-suite for ten times.

Project	EnFuzz$^-$	EnFuzz-A	EnFuzz-Q	EnFuzz-L	EnFuzz
boringssl	3210	3996	4013	4016	**4108**
c-ares	**285**	**285**	**285**	**285**	**285**
guetzli	2074	3316	3246	3531	**3644**
lcms	2872	4054	4152	4098	**4169**
libarchive	6092	12689	11793	13267	**13949**
libssh	613	614	**640**	614	614
libxml2	14428	17657	16932	21664	**21899**
openssl-1.0.1	3612	4194	4204	4538	**4673**
openssl-1.0.2	4037	4176	**4292**	4202	4216
openssl-1.1.0	8642	9371	9401	9680	**9827**
pcre2	32471	51801	52751	52267	**53912**
proj4	267	267	267	**907**	**907**
re2	16300	18070	18376	19323	**19688**
woff2	120	120	121	3939	**3945**
freetype2	49927	55952	54193	58018	**58192**
harfbuzz	33915	43301	43379	44419	**44708**
json	4918	7109	7146	7268	**7339**
libjpeg	9826	15997	15387	16984	**17071**
libpng	3816	4487	4502	4589	**4696**
llvm	49186	58681	58329	60104	**62918**
openthread	2739	3221	4015	5503	**5579**
sqlite	2318	2898	2971	3189	**3216**
vorbis	10328	13872	13993	14210	**14318**
wpantund	33749	41537	41663	43104	**43217**
Total	295745	377665	376051	399719	**407090**
Improvement	–	27% ↑	28% ↑	35% ↑	**38%** ↑

The reason for performance degradation between experiments on LAVA-M and Google fuzzer-test-suite is that the

Table 11: Average number of bugs found by each Enfuzz on Google's fuzzer-test-suite for ten times.

Project	EnFuzz⁻	EnFuzz-A	EnFuzz-Q	EnFuzz-L	EnFuzz
boringssl	0	0	0	**1**	**1**
c-ares	1	**3**	2	**3**	**3**
guetzli	0	0	**1**	**1**	**1**
lcms	0	1	1	**2**	**2**
libarchive	0	0	**1**	**1**	**1**
libssh	0	0	**2**	**2**	**2**
libxml2	1	1	1	2	**3**
openssl-1.0.1	0	3	3	**4**	**4**
openssl-1.0.2	3	5	5	5	**6**
openssl-1.1.0	2	5	5	**6**	**6**
pcre2	3	6	6	7	**8**
proj4	0	2	2	2	**3**
re2	0	1	1	2	**2**
woff2	0	**1**	**1**	**1**	**1**
freetype2	0	0	0	0	**0**
harfbuzz	0	**1**	**1**	**1**	**1**
json	1	2	2	2	**3**
libjpeg	0	0	0	0	**0**
libpng	0	0	0	0	**0**
llvm	0	1	1	**2**	**2**
openthread	0	0	1	3	**4**
sqlite	0	1	1	2	**3**
vorbis	1	**4**	**4**	**4**	**4**
wpantund	0	0	0	0	0
Total	12	37	41	53	**60**
Improvement	–	208% ↑	242% ↑	342% ↑	**400%** ↑

base codes of the four applications (who, uniq, base64 and md5sum) in LAVA-M are small (2K-4K LOCs). The concolic execution engine works well on them, but usually performs the opposite or even hangs on real projects in fuzzer-test-suite whose code base easily reaches 100k LOCs.

For the EnFuzz-L which integrates AFL, AFLFast, FairFuzz and libFuzzer as base fuzzers, the results are presented in the seventh columns of Tables 9, 10 and 11. As mentioned in section A, the diversity among these base fuzzers is much larger than with EnFuzz-A. Compared with EnFuzz-A, EnFuzz-L always performs better on all target applications. In total, it covers 23.6% more paths, executes 5.8% more branches and triggers 42.4% more unique bugs than EnFuzz-A.

For the EnFuzz which integrates AFL, AFLFast, libFuzzer and Radamsa as base fuzzers, the diversity is the largest because they cover all three diversity heuristics. Compared with EnFuzz-L, it performs better and covers 3.6% more paths, executes 1.8% more branches and triggers 13.2% more unique bugs. Both EnFuzz and EnFuzz-L performs better than EnFuzz-Q. These statistics demonstrate that the more diversity among these base fuzzers, the better the ensemble fuzzer should perform. For real applications with a large code base, compared with hybrid conclic fuzzing or ensemble fuzzing with symbolic execution, the ensemble fuzzing without symbolic execution may perform better.

5.6 Fuzzing Real-World Applications

We apply EnFuzz to fuzz more real-world applications from GitHub and commercial products from Cisco, some of which are well-fuzzed projects such as the image processing library libpng and libjepg, the video processing library libwav, the IoT device communication protocol libiec61850 used in hundreds of thousands of cameras, etc. EnFuzz also performs well. Within 24 hours, besides the coverage improvements, EnFuzz finds 60 more unknown real bugs including 44 successfully registered as CVEs, as shown in Table 13. All of these new bugs and security vulnerabilities are detected in a 64-bit machine with 36 cores (Intel(R) Xeon(R) CPU E5-2630 v3@2.40GHz), 128GB of main memory, and Ubuntu 16.04 as the host OS.

Table 12: Unique previously unknown bugs detected by each tool within 24 hours on some real-world applications.

Project	AFL	AFLFast	FairFuzz	LibFuzzer	QSYM	EnFuzz
Bento4_mp4com	5	4	5	5	4	**6**
Bento4_mp4tag	5	4	4	5	4	**7**
bitmap	1	1	1	0	1	**2**
cmft	1	1	0	1	0	**2**
ffjpeg	1	1	1	0	1	**2**
flif	1	1	1	2	1	**3**
imageworsener	**1**	0	0	0	**1**	**1**
libjpeg-05-2018	3	3	3	4	3	**5**
libiec61850	3	2	2	1	2	**4**
libpng-1.6.34	2	1	1	1	2	**3**
libwav_wavgain	3	2	3	0	2	**5**
libwav_wavinfo	2	1	2	4	2	**5**
LuPng	1	1	1	3	1	**4**
pbc	5	5	6	7	6	**9**
pngwriter	1	1	1	1	**2**	**2**
total	35	28	31	34	32	**60**

As a comparison, we also run each tool on those real-world applications to detect unknown vulnerabilities. The results are presented in table 12. EnFuzz found all 60 unique bugs, while other tools only found a portion of these bugs. Compared with AFL, AFLFast, FairFuzz, LibFuzzer and QSYM, EnFuzz detected 71.4%, 114%, 93.5%, 76.4%, 87.5% more unique bugs respectively. The results demonstrate the effectiveness of EnFuzz in detecting real vulnerabilities in more general projects. For example, in the well-fuzzed projects libwav and libpng, we can still detect 13 more real bugs, 7 of which are assigned as CVEs. We give an analysis of the project libpng for a more detailed illustration. libpng is a widely used C library for reading and writing PNG image files. It has been fuzzed many times and is one of the projects in Google's OSS-Fuzz, which means it has been continually fuzzed by multiple fuzzers many times. But with EnFuzz, we detect three vulnerabilities, including one segmentation fault, one stack-buffer-overflow and one memory leak. The first two vulnerabilities were assigned as CVEs (CVE-2018-14047, CVE-2018-14550).

In particular, CVE-2018-14047 allows remote attackers to cause a segmentation fault via a crafted input. We analyze the vulnerability with AddressSanitizer and find it is a typical memory access violation. The problem is that in function png_free_data in line 564 of png.c, the info_ptr attempts to access an invalid area of memory. The error occurs in png_free_data during the free of text-related data with specifically crafted files, and causes reading of invalid or unknown memory, as show in Listing 1. The new vulnera-

bilities and CVEs in the IoT device communication protocol libiec6185 can also crash the service and have already been confirmed and repaired.

```
#ifdef PNG_TEXT_SUPPORTED
/* Free text item num or (if num ==
    -1) all text items */
     if (info_ptr->text != NULL &&
         ((mask & PNG_FREE_TEXT) &
    info_ptr->free_me) != 0)
```

Listing 1: The error code of libpng for CVE-2018-14047

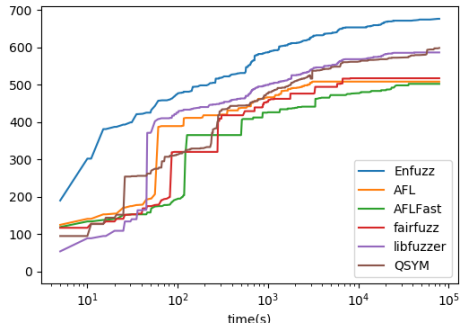
We also apply each base fuzzer (AFL, AFLFast, FairFuzz, libFuzzer and QSYM) to fuzz libpng separately, the above vulnerability is not detected. To trigger this bug, 6 function calls and 11 compares (2 for integer, 1 for boolean and 8 for pointer) are required. It is difficult for other fuzzers to detect bugs in such deep paths without the seeds synchronization of EnFuzz. The performances of these fuzzers over time in libpng are presented in Figure 4. The results demonstrate that generalization and scalability limitations exist in these base fuzzers – the two optimized fuzzers AFLFast and FairFuzz perform worse than the original AFL for libpng, while `EnFuzz` performs the best. Furthermore, except for those evaluations on benchmarks and real projects, `EnFuzz` had already been deployed in industry practice, and more new CVEs were being continuously reported.

Table 13: The 44 CVEs detected by `EnFuzz` in 24 hours.

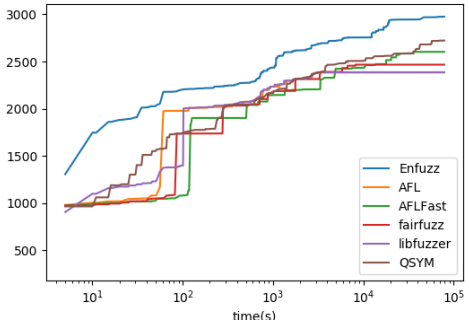
Project	Count	CVE-2018-Number
Bento4_mp4com	6	14584, 14585, 14586, 14587, 14588, 14589
Bento4_mp4tag	6	13846, 13847, 13848, 14590, 14531, 14532
bitmap	1	17073
cmft	1	13833
ffjpeg	1	16781
flif	1	12109
imageworsener	1	16782
libjpeg-05-2018	4	11212, 11213, 11214, 11813
libiec61850	3	18834, 18937, 19093
libpng-1.6.34	2	14048, 14550
libwav_wavgain	2	14052, 14549
libwav_wavinfo	3	14049, 14050, 14051
LuPng	3	18581, 18582, 18583
pbc	9	14736, 14737, 14738, 14739, 14740, 14741, 14742, 14743, 14744
pngwriter	1	14047

6 Discussion

Based on benchmarks such as LAVA-M and Google's fuzzer-test-suite, and several real projects, we demonstrate that this ensemble fuzzing approach outperforms any base fuzzers. However, some limitations still threaten the performance of



(a) Number of paths over time



(b) Number of branches over time

Figure 4: Performance of each fuzzer over time in libpng. Each fuzzer runs in four CPU cores for 24 hours.

ensemble fuzzing. The representative limitations and the workarounds are discussed below.

The first potential threat is the insufficient and imprecise diversity of base fuzzers. Section 4.1 describes our base fuzzer selection, we propose three different heuristics to indicate diversity of base fuzzers, including diversity of coverage information granularity, diversity of input generation strategy, and diversity of seed mutation selection strategy. According to these three heuristics, we select AFL, AFLFast, FairFuzz, libFuzzer, Radamsa and QSYM as the base fuzzers. Furthermore, we implement four prototypes of ensemble fuzzing and demonstrate that the greater the diversity of base fuzzers, the better the ensemble fuzzer performs. However, these three different heuristics of diversity may be insufficient. More diversity measures need to be proposed in future work. For example, initial seeds determine the initial direction of fuzzing and, thus, are significantly important for fuzzing, especially for mutation-based fuzzers. Some fuzzers utilize initial seeds generated by symbolic execution [35, 29] while some other fuzzers utilize initial seeds constructed by domain experts or grammar specifications. However, we select base fuzzers manually according to the initial diversity heuristic, which is also not accurate enough.

A possible solution to this threat is to quantify the initial diversity value among different fuzzers for more accurate selection. As defined in [14], the variance or diversity is a measure of the distance of the data in relation to the average. The average standard deviation of a data set is a percentage that

indicates how much, on average, each measurement differs from the other. To evaluate the diversity of different base fuzzers, we can choose the most widely used AFL and its path coverage as a baseline and then calculate standard deviation of each tool from this baseline on the Google fuzzing-test-suite. Then we can calculate the standard deviation of these values as the initial measure of diversity for each base fuzzer, as presented in formula (2) and (1), where n means the number of applications fuzzed by these base fuzzers, p_i means the number of paths covered by the current fuzzer of the target application i and p_{A_i} means the number of paths covered by AFL of the application i.

$$mean = \frac{1}{n}\sum_{i=1}^{n}\frac{p_i - p_{A_i}}{p_{A_i}} \tag{1}$$

$$diversity = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{p_i - p_{A_i}}{p_{A_i}} - mean\right)^2 \tag{2}$$

Take the diversity of AFLFast, FairFuzz, Radamsa, QSYM, and libFuzzer for example, as shown in the statistics presented in Table 14 of the appendix, compared with AFL on different applications, the diversity of AFLFast is 0.040; the diversity of FairFuzz is 0.062; the diversity of Radamsa is 0.197; the diversity of QSYM is 0.271; the diversity of libFuzzer is 11.929. In the same way, the deviation on branches covered and the bugs detected can be calculated. We can add these three values together with different weight for the final diversity quantification. For example, the bug deviation should be assigned with more weights, because from prior research, coverage metrics (the number of paths or branches) are not necessarily correlated well with bugs found. A more advanced way to evaluate the amount of diversity would be to count how many paths/branches/bugs were found by one fuzzer and not by any of the others.

The second potential threat is the mechanism scalability of the ensemble architecture. Section 4.2 describes the ensemble architecture design, and proposes the globally asynchronous and locally synchronous based seed synchronization mechanism. The seed synchronization mechanism focuses on enhancing cooperation among these base fuzzers during their fuzzing processes. With the help of seeds sharing, the performance of ensemble fuzzing is much improved and is better than any of the constituent base fuzzers with the same computing resources usage. However, this mechanism can still be improved for better scalability on different applications and fuzzing tasks. `EnFuzz` only synchronizes the coarse-grained information – interesting seeds, rather than the fine-grained information. For example, we could synchronize the execution trace and array index values of each base fuzzer to improve their effectiveness in cooperation. Furthermore, we currently select and mix base fuzzers manually according to three heuristics. When scaled to arbitrary number of cores, it should be carefully investigated with huge number of empirical evaluations. A possible solution is that the base fuzzers will be dynamically selected and initiated with different number of cores according to the real-time number of paths/branches/bugs found individually by each fuzzer. In the beginning, we have a set of different base fuzzers; then Enfuzz selects n (this number can be configured) base fuzzers randomly. If one fuzzer cannot contribute to coverage for a long time, then it will be terminated, and one new base fuzzer from the sets will be setup for fuzzing or the existing live base fuzzer with better coverage will be allocated with more CPU cores.

We can also apply some effective ensemble mechanisms in ensemble learning such as Boosting to ensemble fuzzing to improve the scalability. Boosting is a widely used ensemble mechanism which will reweigh the base learner dynamically to improve the performance of the ensemble learner: examples that are misclassified gain weight and examples that are classified correctly lose weight. To implement this idea in ensemble fuzzing, we could start up a *master* thread to monitor the execution statuses of all base fuzzers and record more precise information of each base fuzzer, then reassign each base fuzzer some interesting seeds accordingly.

For the number of base fuzzers and parameters in ensemble fuzzing implementation, it is scalable for integration of most fuzzers. Theoretically, the more base fuzzers with diversity, the better ensemble fuzzing performs. We only use four base fuzzers in our evaluation with four CPU cores. The more computing resources we get, higher performance the fuzzing practice acquires. Furthermore, in our implementation, we have tried different values of period time, and the results are very sensitive to the specific setting of this value. It only affects the performance in the beginning, but affects little in the end. Furthermore, refering to the GALS system design, we can also allocate a different synchronization frequency for each local fuzzer dynamically.

7 Conclusion

In this paper, we systematically investigate the practical ensemble fuzzing strategies and the effectiveness of ensemble fuzzing of various fuzzers. Applying the idea of ensemble fuzzing, we bridge two gaps. First, we come up with a method for defining the diversity of base fuzzers and propose a way of selecting a diverse set of base fuzzers. Then, inspired by AFL in parallel mode, we implement a concrete ensemble architecture with one effective ensemble mechanism, a seed synchronization mechanism. `EnFuzz` always outperforms other popular base fuzzers in terms of unique bugs, path and branch coverage with the same resource usage. `EnFuzz` has found 60 new bugs in several well-fuzzed projects and 44 new CVEs were assigned. Our ensemble architecture can be easily utilized to integrate other base fuzzers for industrial practice.

Our future work will focus on three directions: the first is to try some other heuristics and more accurate accumulated quantification of diversity in base fuzzers; the second is to improve the ensemble architecture with more advanced ensemble mechanism and synchronize more fine-grained information; the last is to improve the ensemble architecture with intelligent resource allocation such as dynamically adjusting the synchronization period for each base fuzzer, and allocating more CPU cores to the base fuzzer that shares more interesting seeds.

References

[1] Fuzzer automation with spike. http://resources.infosecinstitute.com/fuzzer-automation-with-spike/. [Online; accessed 12-February-2018].

[2] Cert bff - basic fuzzing framework. https://vuls.cert.org/confluence/display/tools/CERT+BFF+-+Basic+Fuzzing+Framework, 2012. [Online; accessed 10-April-2018].

[3] Afl in parallel mode. https://github.com/mcarpenter/afl/blob/master/docs/parallel_fuzzing.txt, 2016. [Online; accessed 10-April-2019].

[4] Continuous fuzzing for open source software. https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html, 2016. [Online; accessed 10-April-2018].

[5] Google. honggfuzz. https://google.github.io/honggfuzz/, 2016. [Online; accessed 10-April-2018].

[6] libfuzzer in parallel mode. https://github.com/google/fuzzer-test-suite/blob/master/tutorial/libFuzzerTutorial.md, 2016. [Online; accessed 10-April-2019].

[7] Technical details for afl. http://lcamtuf.coredump.cx/afl/technical_details.txt, 2016. [Online; accessed 10-April-2019].

[8] fuzzer-test-suite. https://github.com/google/fuzzer-test-suite, 2017. [Online; accessed 10-April-2018].

[9] Google security blog. https://security.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html, 2017. [Online; accessed 10-April-2018].

[10] libfuzzer. https://llvm.org/docs/LibFuzzer.html, 2017. [Online; accessed 10-April-2018].

[11] Sanitizercoverage in llvm. https://clang.llvm.org/docs/SanitizerCoverage.html, 2017. [Online; accessed 10-April-2018].

[12] Clusterfuzz document. https://github.com/google/oss-fuzz/blob/master/docs/clusterfuzz.md, 2018. [Online; accessed 2-November-2018].

[13] Clusterfuzz integration document. https://chromium.googlesource.com/chromium/src/testing/libfuzzer/+/HEAD/clusterfuzz.md, 2018. [Online; accessed 2-November-2018].

[14] BENJAMIN, J. R., AND CORNELL, C. A. *Probability, statistics, and decision for civil engineers*. Courier Corporation, 2014.

[15] BÖHME, M., PHAM, V.-T., NGUYEN, M.-D., AND ROYCHOUDHURY, A. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS17)* (2017).

[16] BÖHME, M., PHAM, V.-T., AND ROYCHOUDHURY, A. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 1032–1043.

[17] CHA, S. K., WOO, M., AND BRUMLEY, D. Program-adaptive mutational fuzzing. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 725–741.

[18] CHEN, P., AND CHEN, H. Angora: Efficient fuzzing by principled search. *arXiv preprint arXiv:1803.01307* (2018).

[19] DOLAN-GAVITT, B., HULIN, P., KIRDA, E., LEEK, T., MAMBRETTI, A., ROBERTSON, W., ULRICH, F., AND WHELAN, R. Lava: Large-scale automated vulnerability addition. In *Security and Privacy (SP), 2016 IEEE Symposium on* (2016), IEEE, pp. 110–121.

[20] EDDINGTON, M. Peach fuzzing platform. *Peach Fuzzer* (2011), 34.

[21] GODEFROID, P., KIEZUN, A., AND LEVIN, M. Y. Grammar-based whitebox fuzzing. In *ACM Sigplan Notices* (2008), vol. 43, ACM, pp. 206–215.

[22] HELIN, A. Radamsa. https://gitlab.com/akihe/radamsa, 2016.

[23] HOCEVAR, S. zzuf - multi-purpose fuzzer. http://caca.zoy.org/wiki/zzuf, 2007. [Online; accessed 10-April-2018].

[24] HOLLER, C., HERZIG, K., AND ZELLER, A. Fuzzing with code fragments. In *USENIX Security Symposium* (2012), pp. 445–458.

[25] KLEES, G., RUEF, A., COOPER, B., WEI, S., AND HICKS, M. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), ACM, pp. 2123–2138.

[26] LEMIEUX, C., AND SEN, K. Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage. *arXiv preprint arXiv:1709.07101* (2017).

[27] LIANG, J., JIANG, Y., CHEN, Y., WANG, M., ZHOU, C., AND 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* (2018), ACM, pp. 809–814.

[28] LIANG, J., WANG, M., CHEN, Y., JIANG, Y., AND ZHANG, R. Fuzz testing in practice: Obstacles and solutions. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2018), IEEE, pp. 562–566.

[29] OGNAWALA, S., HUTZELMANN, T., PSALLIDA, E., AND PRETSCHNER, A. Improving function coverage with munch: a hybrid fuzzing and directed symbolic execution approach. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing* (2018), ACM, pp. 1475–1482.

[30] PETSIOS, T., ZHAO, J., KEROMYTIS, A. D., AND JANA, S. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 2155–2168.

[31] SIRER, E. G., AND BERSHAD, B. N. Using production grammars in software testing. In *ACM SIGPLAN Notices* (1999), vol. 35, ACM, pp. 1–13.

[32] STEPHENS, N., GROSEN, J., SALLS, C., DUTCHER, A., WANG, R., CORBETTA, J., SHOSHITAISHVILI, Y., KRUEGEL, C., AND VIGNA, G. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS* (2016), vol. 16, pp. 1–16.

[33] VEGGALAM, S., RAWAT, S., HALLER, I., AND BOS, H. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security* (2016), Springer, pp. 581–601.

[34] WANG, J., CHEN, B., WEI, L., AND LIU, Y. Skyfire: Data-driven seed generation for fuzzing, 2017.

[35] WANG, M., LIANG, J., CHEN, Y., JIANG, Y., JIAO, X., LIU, H., ZHAO, X., AND SUN, J. Safl: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (2018), ACM, pp. 61–64.

[36] XU, W., KASHYAP, S., MIN, C., AND KIM, T. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 2313–2328.

[37] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. Finding and understanding bugs in c compilers. In *ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 283–294.

[38] YUN, I., LEE, S., XU, M., JANG, Y., AND KIM, T. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)* (2018), pp. 745–761.

[39] ZALEWSKI, M. American fuzzy lop. `https://github.com/mcarpenter/afl`, 2015.

A Preliminary demonstration of diversity among base fuzzers

To help select base fuzzers with larger diversity, we need to estimate the diversity between each base fuzzer. In general, the more differently they perform on different applications, the more diversity among these base fuzzers. Accordingly, we first run each base fuzzer in single mode, with one CPU core on Google's fuzzer-test-suite for 24 hours. Table 14 and Table 15 show the number of paths and branches covered by AFL, AFLFast, FairFuzz, libFuzzer, Radamsa and QSYM. Table 16 shows the corresponding number of unique bugs. Below we present the performance effects of the three diversity heuristics proposed in Section 4.1 in detail.

1) Effects of seed mutation and seed selection strategy – what kind of mutation and selection strategy you use, what kind of path and branch you would cover The first three columns of Table 14 show the performance of the AFL family tools. Their differences are the seed mutation and seed selection strategies. The original AFL performs the best on 5 applications, but performs the worst on other 10 applications. AFLFast performs the best on 13 applications, and only performs the worst on 4 applications. FairFuzz also performs the best on 8 applications, but the worst on the other 9 applications. Although the total number of paths covered improves slightly, the performance variation on each application is huge, ranging from -57% to 38% in single cases.

From the first three columns in Table 15 and Table 16, we get the same observation that the performance of these optimized fuzzers varies significantly on different applications. Although the total number of covered branches and unique crashes improves slightly, the deviation of each application is huge. AFLFast selects seeds that exercise low-frequency paths to mutate more times. Take project lcms for example, this seed selection strategy exercises more new paths by avoiding covering "hot paths" too many times, but on project libarchive, its "hot path" may be the key to further paths. FairFuzz mutates seeds to hit rare branches. Take project libxml2 for example, the rare branch fuzzing strategy guides FairFuzz into deeper areas and covers more branches. However, on libarchive, this strategy fails. FairFuzz spends much time in deep paths and branches, ignoring breadth search. Unlike libxml2, the breadth first search strategy of other fuzzers is more effective on libarchive. In general, the mutation and selection strategy decides the depth and breath of the covered branch and path.

2) Effects of coverage information granularity–what kind of guided information you use, what kind of coverage metric you improve. The diversity between AFL and libFuzzer is their coverage information granularity. According to the fourth column of Table 14, we find that compared with AFL, libFuzzer performs better on 17 applications, and covers 30.3% more paths in total. However, according to the fourth column of the Table 15, compared with AFL, libFuzzer only performs better on 11 applications, which means on 6 applications, libFuzzer covers more paths but less branches. For

total branch count, AFL covers 7.3% more than libFuzzer. The reason is that AFL mutates seed by tracking edge hit counts while libFuzzer utilizes the SanitizerCoverage instrumentation to track block hit counts. AFL prefers to cover more branches while libFuzzer is better at executing more paths. In general, edge-guided means more branches covered, and block-guided means more paths covered.

Table 14: Average number of paths for single mode.

Project	AFL	AFLFast	FairFuzz	libFuzzer	Radamsa	QSYM
boringssl	1334	1674	1760	**3528**	1682	1207
c-ares	80	84	88	**123**	78	72
guetzli	1382	1090	1030	**1773**	1562	1268
lcms	656	**864**	434	338	550	605
libarchive	3756	2834	1630	**10124**	4570	3505
libssh	64	68	62	**201**	63	87
libxml2	5762	7956	8028	**19663**	9392	5098
openssl-1.0.1	**2397**	2103	2285	1709	2303	2330
openssl-1.0.2	2456	**2482**	2040	1881	2108	1947
openssl-1.1.0	2439	2380	**2501**	1897	2311	2416
pcre2	32310	35288	36176	20981	**37850**	24501
proj4	220	218	218	**334**	182	208
re2	5860	6014	5016	**6327**	5418	5084
woff2	14	10	12	**224**	10	15
freetype2	7748	10939	10714	**16360**	9825	7188
harfbuzz	6793	8068	8668	**10800**	5688	6881
json	466	412	408	499	**564**	504
libjpeg	704	**979**	722	448	634	638
libpng	170	159	76	263	493	**577**
llvm	4830	**5760**	5360	5646	4593	4096
openthread	104	123	127	**976**	144	141
sqlite	179	193	172	**431**	256	180
vorbis	891	**1122**	821	848	875	898
wpantund	2959	3048	**3513**	3510	3146	2975
Total	83575	93867	91862	**108884**	94296	72422

Table 15: Average number of branches for single mode.

Project	AFL	AFLFast	FairFuzz	libFuzzer	Radamsa	QSYM
boringssl	2645	3054	3115	3608	**3641**	2539
c-ares	**126**	122	**126**	100	**126**	**126**
guetzli	1913	1491	1428	**2774**	2118	1906
lcms	2216	**2755**	935	2661	1661	2075
libarchive	4906	3961	2387	3561	**5263**	4366
libssh	604	604	604	518	604	**626**
libxml2	10082	12407	12655	13037	**14287**	9779
openssl-1.0.1	3809	3879	**3901**	2591	2993	3829
openssl-1.0.2	3978	4015	3883	2308	**4068**	3796
openssl-1.1.0	8091	8132	8212	7810	**8292**	8032
pcre2	27308	29324	28404	13463	**30615**	19557
proj4	264	260	260	**683**	264	258
re2	15892	15970	15073	11369	**16485**	14477
woff2	114	112	114	**1003**	114	115
freetype2	36798	44028	45319	45541	**49468**	33492
harfbuzz	16872	16051	**19045**	18659	16782	16886
json	4462	3626	**4846**	4547	4821	4538
libjpeg	6865	8495	4028	**8828**	6982	6377
libpng	1917	1878	1135	1651	2126	**2294**
llvm	54107	55697	**57356**	51548	53427	47226
openthread	2062	2473	2646	**5295**	2231	2410
sqlite	2706	**2784**	2771	2178	2190	2709
vorbis	11836	**13561**	12605	5902	11217	12531
wpantund	36059	36620	**37269**	28694	37075	35960
Total	255631	271299	268116	238329	**276850**	235903

3) Effects of Input generation strategy–what kind of generation strategy you use, what kind of corresponding application you fuzz better. The diversity between AFL and Radamsa is the input generation strategy. From the fifth columns of Table 14 and Table 15, compared with AFL, the plenty of inputs generated by Radamsa have some side effects on most target applications (14 applications). Too many extra inputs will slow down the execution speed of the fuzzer. However, for some applications, the inputs generated by Radamsa will improve the performance effectively. Take libxml2 for example, Radamsa has some domain knowledge that prefers to generate some structured data and specific complex format data. These domain knowledge are not available in most mutation-based fuzzers, and this is a critical disadvantage of AFL. But with the help of generation-based fuzzers, the performance of AFL can be improved greatly.

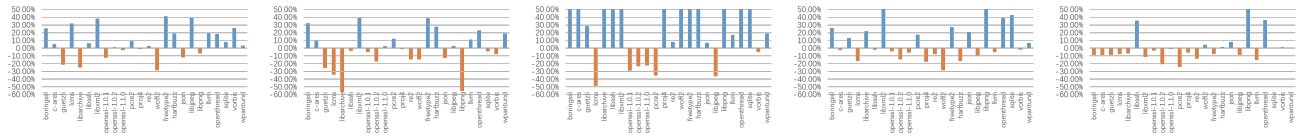
Table 16: Average number of bugs for single mode.

Project	AFL	AFLFast	FairFuzz	libFuzzer	Radamsa	QSYM
boringssl	0	0	0	**1**	0	0
c-ares	1	**2**	**2**	1	**2**	1
guetzli	0	0	0	0	0	0
lcms	0	0	0	0	0	0
libarchive	0	0	0	0	0	0
libssh	0	0	0	**1**	0	0
libxml2	0	**1**	0	**1**	**1**	0
openssl-1.0.1	0	0	0	0	0	0
openssl-1.0.2	**2**	1	0	1	1	**2**
openssl-1.1.0	0	0	0	0	0	0
pcre2	**2**	1	1	1	**2**	1
proj4	0	0	0	**1**	0	0
re2	0	0	0	**1**	0	0
woff2	0	0	0	**1**	0	0
freetype2	0	0	0	0	0	0
harfbuzz	0	0	0	**1**	0	0
json	**1**	**1**	0	0	**1**	0
libjpeg	0	0	0	0	0	0
libpng	0	**1**	**1**	**1**	**1**	**1**
llvm	0	0	**1**	**1**	0	**1**
openthread	0	0	0	**1**	0	0
sqlite	0	0	0	**1**	**1**	**1**
vorbis	1	1	**2**	1	1	**2**
wpantund	0	0	0	0	0	0
Total	7	8	7	**15**	10	9

In conclusion: Different base fuzzers perform variously on distinct target applications, showing the diversity for the base fuzzers. The more diversity of these base fuzzers, the more differently they perform on different applications. Furthermore, the above three types of effects should be considered and could be incorporated into the fuzzing evaluation guideline [25] to avoid biased test cases or metrics selection when evaluating different types of fuzzing optimization.

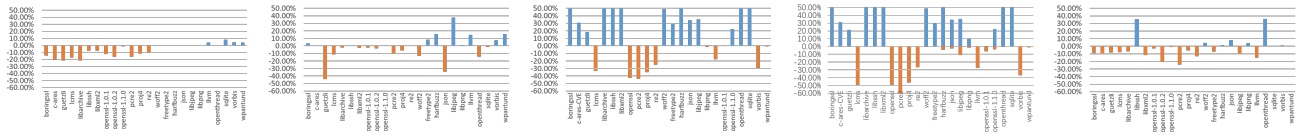
B Does performance vary in different modes?

We choose AFL as the baseline, and compare other tools with AFL on path coverage to demonstrate the performance variation. Figure 5 shows the average number of paths executed on Google's fuzzer-test-suite by each base fuzzer compared with AFL in single mode. We also collect the result of



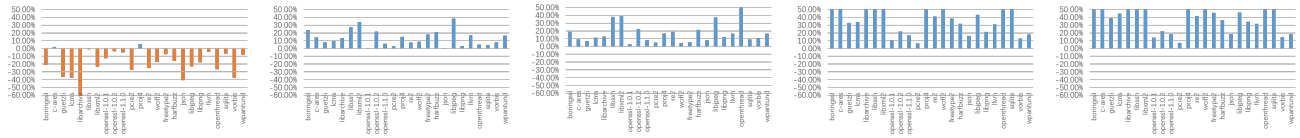
(a) performance of AFLFast in single thread (b) performance of FairFuzz in single thread (c) performance of libFuzzer in single thread (d) performance of Radamsa in single thread (d) performance of QSYM in single thread

Figure 5: Paths covered by base fuzzers compared with AFL in single mode on a single core.



(a) performance of AFLFast in four threads (b) performance of FairFuzz in four threads (c) performance of libFuzzer in four threads (d) performance of Radamsa in four threads (d) performance of QSYM in four threads

Figure 6: Paths covered by base fuzzers compared with AFL in parallel mode with four threads on four cores.



(a) performance of EnFuzz$^-$ in four threads (a) performance of EnFuzz-A in four threads (a) performance of EnFuzz-Q in four threads (b) performance of EnFuzz-L in four threads (c) performance of EnFuzz in four threads

Figure 7: Paths covered by EnFuzz with four threads on four cores compared with AFL in parallel mode with four threads on four cores. EnFuzz$^-$ without the proposed seed synchronization performs the worst, and EnFuzz performs the best.

each base fuzzer running in parallel mode with four threads, and the result is presented in Figure 6. Figure 7 shows the average number of paths executed by EnFuzz compared with AFL in parallel mode with four CPU cores. From these results, we get the following conclusions:

- From the results of Figure 5 and Figure 6, we find that compared with AFL, the two optimized fuzzers AFLFast and FairFuzz, block coverage guided fuzzer libFuzzer, generation-based fuzzer Radamsa and hybrid fuzzer QSYM perform variously on different applications both in single mode and in parallel mode. It demonstrates that the performance of these base fuzzers is challenged by the diversity of the diverse real applications. The performance of their fuzzing strategies cannot constantly perform better than AFL. The performance variation exists in these state-of-the-art fuzzers.

- Comparing the result of Figure 5 and Figure 6, we find that the performance of these base fuzzers in parallel mode are quite different from those in single mode, especially for AFLFast and FairFuzz. In single mode, the other two optimized base fuzzers perform better than AFL in many applications. But in parallel mode, the result is completely opposite that the original AFL performs better on almost all applications.

- From the result of Figure 7, it reveals that EnFuzz-A, EnFuzz-L and EnFuzz always perform better than AFL on the target applications. For the same computing resources usage where AFL running in parallel mode with four CPU cores, EnFuzz-A covers 11.26% more paths than AFL, ranging from 4% to 38% in single cases, EnFuzz-Q covers 12.48% more paths than AFL, ranging from 5% to 177% in single cases, EnFuzz-L covers 37.50% more paths than AFL, ranging from 13% to 455% in single cases. EnFuzz covers 42.39% more paths than AFL, ranging from 14% to 462% in single cases. Through ensemble fuzzing, the performance variation can be reduced.

- From the result of Figure 7, it reveals that EnFuzz$^-$ without seed synchronization performs worse than AFL parallel mode under the same resource constraint. Compared with EnFuzz-A, EnFuzz-Q covers 1.09% more paths, EnFuzz-L covers 23.58% more paths. For EnFuzz, it covers 27.97% more paths than EnFuzz-A, 26.59% more paths than EnFuzz-Q, 3.6% more paths than EnFuzz-L, and always performs the best on all applications. The more diversity among those integrated base fuzzers, the better performance of ensemble fuzzing, and the seed synchronization contributes more to the improvements.

In conclusion: the performance of the state-of-the-art fuzzers is greatly challenged by the diversity of those real-world applications, and it can be improved through the ensemble fuzzing approach. Furthermore, those optimized strategies work in single mode can not be directly scaled to parallel mode which is widely used in industrial practice. The ensemble fuzzing approach is a critical enhancement to the single and parallel mode of those optimized strategies.