EnFuzz: From Ensemble Learning to Ensemble Fuzzing

Yuanliang Chen*, Yu Jiang*, Jie Liang*, Mingzhe Wang* and Xun Jiao*
* School of Software, Tsinghua University, KLISS, Beijing, China
Email: chenyuan17@mails.tsinghua.edu.cn, {liangjie.mailbox.cn, wmzhere}@gmail.com,
jiangyu198964@126.com, canbyjiaoxun@163.com

***Abstract*—Fuzzing is widely used for software vulnerability detection. There are various kinds of fuzzers with different fuzzing strategies, and most of them perform well on their targets. However, in industry practice and empirical study, the performance and generalization ability of those well-designed fuzzing strategies are challenged by the complexity and diversity of real-world applications.**

In this paper, inspired by the idea of ensemble learning, we first propose an ensemble fuzzing approach `EnFuzz`, that integrates multiple fuzzing strategies to obtain better performance and generalization ability than that of any constituent fuzzer alone. First, we define the diversity of the base fuzzers and choose those most recent and well-designed fuzzers as base fuzzers. Then, `EnFuzz` ensembles those base fuzzers with seed synchronization and result integration mechanisms. For evaluation, we implement `EnFuzz`, a prototype basing on four strong open-source fuzzers (AFL, AFLFast, AFLGo, FairFuzz), and test them on Google's fuzzing test suite, which consists of widely used real-world applications. The 24-hour experiment indicates that, with the same resources usage, these four base fuzzers perform variously on different applications, while `EnFuzz` shows better generalization ability and always outperforms others in terms of path coverage, branch coverage and crash discovery. Even compared with the best cases of AFL, AFLFast, AFLGo and FairFuzz, `EnFuzz` discovers 26.8%, 117%, 38.8% and 39.5% more unique crashes, executes 9.16%, 39.2%, 19.9% and 20.0% more paths and covers 5.96%, 12.0%, 21.4% and 11.1% more branches respectively.

***Index Terms*—Software Testing, Ensemble Fuzzing**

I. Introduction

Fuzzing is one of the most popular software testing techniques for bug and vulnerability detection, and there are many fuzzers for academic and industry usage. The key idea is to generate plenty of inputs to execute the target application and monitor anomalies. Each fuzzer develops its own unique fuzzing strategy to generate inputs. There are mainly two types of strategies. One is generation-based strategy that uses the specification of input format, e.g. grammar, to generate complex inputs. For example, IFuzzer [31] takes a context-free grammar as specification to generate parse trees for code fragments. The other is mutation-based strategy that generates new inputs by mutating the existing inputs. Recently, mutation-based fuzzers has adopt coverage information to further improve efficiency. For example, libFuzzer [7] mutates inputs by utilizing the SanitizerCoverage [8] instrumentation, which supports tracking block coverage; while AFL [36] uses static instrumentation with a bitmap to track edge coverage.

Basing on these two underlying fuzzing strategies, researchers have done lots of optimizations. For example, AFLFast [11] improves the fuzzing strategy of AFL by selecting seeds exercising low-frequency paths to mutate more times, and FairFuzz [25] optimizes AFL's mutation algorithm to prioritize rare branches. AFLGo [10] assigns more mutation times to the seeds closer to target locations. All these optimized fuzzers perform well on some target applications and they have already identified a large number of software bugs and security vulnerabilities.

However, when we apply these optimized fuzzers on some real-world applications, these fuzzing strategies can not always perform well, and the efficiency on different applications varies accordingly. For example, on some target applications, FairFuzz or AFLFast may perform better than the other fuzzers, while on some other applications, the original AFL performs the best. The generalization ability of existing fuzzers is challenged by complexity and diversity of real-world applications. For any given real-world application, we can not decide which fuzzer is better unless we spend lots of time analyzing or running all these fuzzers one by one, which would waste a lot of human resources and computing resources. Therefore, we need a fuzzer with better generalization ability to provide more stable fuzzing performance.

The theory of ensemble learning [37] proves that the generalization ability of an ensemble learner is usually much stronger than that of base learners [20], [27]. Similarly, we propose the idea of ensemble fuzzing, which intuitively helps in the two following aspects: *robustness*, i.e. the consistent advantage on any application in the evaluation setup; *performance*, i.e. achieving better metrics than any other fuzzers with the same resources.

To demonstrate the effectiveness of ensemble fuzzing, we need to deal with two main challenges before implementation:

1. *Base Fuzzer Selection.* Base fuzzers are the underlying basics, and the diversity of based fuzzers is crucial to an ensemble fuzzer. The greater the diversity of these base fuzzers, the better the ensemble fuzzer perform.
2. *Ensemble Architecture Design.* The architecture directly determines the efficiency of the performance. The concrete ensemble architecture should be carefully designed, because a good ensemble method efficiently combines these existing base fuzzers together into a stronger ensemble fuzzer.

In this paper, to our best knowledge, we first propose an ensemble fuzzing approach EnFuzz. First, we define the diversity of base fuzzers focusing on three dimensions: coverage information granularity diversity, seed generation strategy diversity, and seed mutation as well as selection strategy diversity. Then, we propose an ensemble architecture with two ensemble mechanisms to integrate these base fuzzers efficiently. The first is the seed synchronization mechanism. To promote cooperation among base fuzzers, this mechanism synchronizes interesting seeds from all fuzzers running on the same target application. The second is the result integration mechanism, which completes the fuzzing session by collecting, deduplicating and triaging results from all base fuzzers.

For evaluation, we implement a prototype of EnFuzz, basing on four high-performance and strategy-diverse base fuzzers, including AFL, AFLFast, AFLGo and FairFuzz. All fuzzers are tested on a widely used benchmark — Google's fuzzer-test-suite consisting of real-world applications. The number of paths executed, branches covered and unique crashes discovered are used as metrics. With the same resources usage, after 24 hours, the results demonstrate that these four base fuzzers perform variously on different applications, while EnFuzz not only effectively alleviate the generalization problem of existing fuzzers, but also improves the fuzzing performance. For example, there are many cases that original AFL performs better than the two optimized fuzzers FairFuzz [25] and AFLFast[11] on those real-world applications. Furthermore, even compared with the best case of AFL, AFLFast, AFLGo and FairFuzz, in total, EnFuzz triggers 26.8%, 117%, 38.8% and 39.5% more unique crashes, executes 9.16%, 39.2%, 19.9% and 20.0% more paths and covers 5.96%, 12.0%, 21.4% and 11.1% more branches respectively. This paper makes the following main contributions:

1. We first propose an ensemble fuzzing approach to solve the generalization limitation of existing fuzzers and implemented a prototype EnFuzz for a more stable and robust fuzzing practice. The prototype is open-source [1] and can be used to integrate other fuzzers.
2. We evaluate EnFuzz on a widely used third-party benchmark consisting of real-world applications. The empirical results reveal the generalization limitations of existing fuzzers and demonstrate that the limitations can be mitigated effectively with EnFuzz.

The rest of this paper is organized as follows: Section II introduces related work of fuzzing and learning. Section III elaborates the idea of ensemble fuzzing, including the base fuzzers selection and ensemble architecture design. Section IV presents the implementation and evaluation of EnFuzz. Section V discusses the potential threats and solution of EnFuzz, and we conclude in section VI.

II. Related Work

We introduce the related work about the generation based fuzzing, mutation based fuzzing, fuzzing in practice, ensemble learning and the main difference of our work.

[1]https://github/XXX Closed for double blind review

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 effectively fuzz the target applications which 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 [17], [32], [1] follow an on-line or off-line model of protocol. Hence, they are able to find more complex bugs by creating complex interactions with the target applications. Peach [17] is one of the most popular model-based fuzzer 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 to cooperate with fuzzing targets. By integrating fuzzing with models of data and state, Peach works effectively. Skyfire [32] first learns a context-sensitive grammar model, and then it generates massive inputs basing on this model. Some other popular fuzzers [19], [35], [28], [31], [22] generate inputs basing on context free grammar. P Godefroid enhances whitebox fuzzing of complex structured-input applications by using symbolic execution [19], which directly generates grammar-based constraints whose satisfiability is checked using a custom grammar-based constraint solver. Csmith [35] is designed for fuzzing C-compilers, and it generates plenty of random C programs in the C99 standard as the inputs. This tool can be used to generate C programs exploring a typical combination of C-language features while free of undefined and unspecified behaviors. LAVA [28] generates effective test suites for the Java virtual machine by specifying production grammars. IFuzzer [31] first constructs parse trees basing on a language's context-free grammar, then it generates new code fragments basing on these parse trees.

2) Mutation-based fuzzing: Mutation-based fuzzers [21], [13], [2] mutate existing test cases to generate new test cases without any input grammar or input model. Traditional mutation-based fuzzers such as zzuf [21] mutates the test cases by flipping random bits, and the mutation ratio of zzuf is predefined. In constrast, the mutation ratio of SYMFUZZ [13] 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 basing on these dependencies. Furthermore, BFF [2] integrates machine learning with evolutionary computing techniques to reassign the mutation ratio. Other popular AFL family tools [36], [11], [10], [25], [33] apply various strategies to boost fuzzing process. AFLFast [11] regards the process of AFL as a Markov chain. A power schedule based on path frequency is responsible for computing the times of random mutation for each seed. Similar to AFLFast, AFLGo [10] also proposes a simulated annealing-based power schedule, which helps to fuzz target code. Especially, it describes a method to measure the distance between the seeds and the targets. FairFuzz [25] mainly focuses on the mutation algorithm. It only mutates seeds which hit rare branches and it strives to ensure the mutant seeds hit the rarest one.

3) Fuzzing in practice: Fuzzing has become the dominant vulnerability discovery solution in industry and has already

found a large number of dangerous bugs and security vulnerabilities in a widespread of systems so far. For example, Google's OSS-Fuzz [7], [3] platform has found more than 1000 bugs in 5 months with thousands of virtual machines [6] by using several popular fuzzers, including libFuzzer, honggfuzz [4] and AFL. In fact, the OSS-Fuzz platform has already embodied the idea of ensemble fuzzing — using multiple different fuzzers together. However, it just uses these fuzzers independently and does not combine them together effectively. In this way, it can not improve the performance of these fuzzers while wasting lots of computing resources.

In industry practice, many existing fuzzers also provide a parallel mode, which allows fuzzing one target application with multiple parallel fuzzers across multiple cores. The parallel mode can effectively improve the performance of fuzzers. The more fuzzing jobs you use, the better they perform. In fact, the parallel mode can be seen as a special method of ensemble fuzzing which uses multiple same base fuzzers. However, the diversity of these base fuzzers is low — all the fuzzing jobs using the same fuzzing strategy with a little randomness. Consequently, only minor performance improvements are observed.

4) Ensemble learning: Ensemble learning is a machine learning paradigm where multiple learners are trained to solve the same problem. In contrast to ordinary machine learning approaches which try to learn one hypothesis from training data, ensemble methods try to construct a set of hypotheses and combine them for precision. The diversity of base learners and the concrete ensemble methods are the two cores of ensemble learning. How to select base learners with great diversity and how to combine these base learners together are critical to the performance of ensemble learners. Many researches focus on these two parts to improve the generalization ability and the prediction accuracy of ensemble learners [27], [9], [30], [30]. A number of diversity measures have been designed in ensemble learning [23], [24]. In addition, there are many effective ensemble methods, such as Boosting [12], Bagging [23], [18] and Stacking [34]. Ensemble learning has achieved a great success in most machine learning areas attributing to better generalization ability.

5) Main Difference: Unlike those mentioned work, we do not propose any new concrete generation- or mutation-based fuzzing strategy, or just run different fuzzers on different machines, but propose a new idea of ensemble fuzzing. To generate a stronger and more stable ensemble fuzzer, we design an ensemble approach consisting of several synchronizing mechanisms to combine those existing efficient base fuzzers together. Furthermore, when fuzzing general real-world applications, a critical difference between ensemble learning and ensemble fuzzing is computing resources usage. For ensemble learning, multiple learners are trained to solve the same problem, which means it uses multiple computing resources. It is insensitive to computing resources usage. But for fuzzing, in general, more computing resources usage means more paths covered and more crashes found. Therefore, different from ensemble learning, it is unfair to compare ensemble fuzzer with any single base fuzzer alone. To solve this problem, we design an evaluation method to compare the ensemble fuzzer with any of the constituent base fuzzer with the same resources usage.

III. Ensemble Fuzzing Approach EnFuzz

The main idea of ensemble fuzzing is constructing a set of base fuzzers and combining them to fuzz the same target application together. The overview of the approach is presented in Figure 1. First of all, when a target application is prepared for fuzzing, we firstly choose some existing fuzzers with diversity as base fuzzers to achieve high generalization ability. The generalization ability of a fuzzer can be described as the ability to perform well, no matter what the target application is. The existing fuzzing strategies of any single fuzzer are usually designed with some preferences, and the performance of these preferences varies greatly on different applications. They can be helpful in some applications, but not certainly effective on other applications. After the base fuzzer selection, we ensemble them in a carefully designed ensemble architecture with two key mechanisms for seed synchronization and result integration. Finally, we collect crash and coverage information as the fuzzing report.

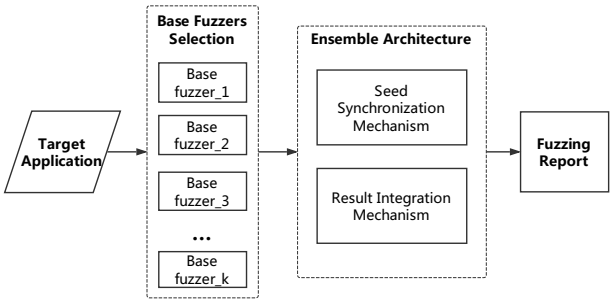


Fig. 1. The overview of ensemble fuzzing consists of base fuzzer selection and ensemble architecture design. The base fuzzers selection contains the diversity definition and quantification, and the architecture design includes seed synchronization and result integration.

A. Base Fuzzers Selection

The first step is to define the base fuzzers. An ensemble fuzzer incorporates any number of existing fuzzers, namely the base fuzzers. These fuzzers can be generation-based fuzzers, e.g. Peach and IFuzzer, or mutation-based fuzzers, e.g. Google's libFuzzer and AFL. We can randomly choose some base fuzzers, but well-defined diversity of base fuzzers and selecting base fuzzers with great diversity contribute to the performance of an ensemble fuzzer.

Let us take the coverage information as an example. Suppose that we have a set of base fuzzers $F = \{fuzzer_1, fuzzer_2, \ldots, fuzzer_k\}$. The code coverage of target program P by $fuzzer_j$ is C_j. If we use all these K base fuzzers together to fuzz the target program P, the total code coverage of P will be their union, that is $C = C_1 \cup C_2 \cup C_3 \ldots \cup C_k$. The following two points are critical:

1. If $C_1 \cap C_2 \cap C_3 \ldots \cap C_k \neq \emptyset$, then $|C| > |C_j|$ where $j \in 1, \ldots, k$, it means that the ensemble fuzzer always performs better than that of any base fuzzer.
2. The smaller $|C_i \cap C_j|$ is, the bigger $|C_i \cup C_j|$ is, and the bigger $|C|$ is, where $0 \leq i, j \leq k, i \neq j$.

Therefore, the greater diversity of these base fuzzers, the better coverage ensemble fuzzers perform.

We define the diversity of base fuzzers basing on three dimensions: seed mutation and selection strategy diversity, coverage information granularity diversity, seed generation strategy diversity. The diversity rules are as follows:

1. Seed mutation and selection strategy based rule. The diversity of base fuzzers can be reflected on the various seed mutation strategies and seed selection strategies. These two strategies are critical to the mutation-based fuzzers. The main difference of most fuzzers are these fuzzing strategies.
2. Coverage information granularity based rule. Lots of base fuzzers determine interesting inputs by tracking different coverage information. Hence, the coverage information of code is critical, and different kinds of coverage granularity tracked by fuzzers promote diversity. For example, libFuzzer guides seed mutation by tracking block coverage while AFL tracks edge coverage — the diversity of them is great.
3. Seed generation strategy based rule. Initial seeds determine the initial direction of fuzzing, thus they are hugely important for fuzzing, especially for mutation-based fuzzers. Therefore, different initial seed generation strategies have critical influences on the diversity of fuzzers. For example, some fuzzers use initial seeds generated by symbolic execution while some other fuzzers use initial seeds constructed by domain experts or grammar specifications.

Based on these three basic rules, we can select base fuzzers as diverse as possible. In ensemble learning, a number of diversity measures have been designed, such as sub-sampling the training examples [26], manipulating the attributes [15], manipulating the outputs [16], injecting randomness into learning algorithms [14], [15], [16], or even using multiple mechanisms simultaneously [29]. These advanced diversity measures can also be customized in ensemble fuzzing basing on these three rules.

B. Ensemble Architecture Design

After choosing base fuzzers with great diversity, the next step is to design a suitable architecture to integrate these base fuzzers together into a stronger ensemble fuzzer. There are two main mechanisms within our proposed architecture. The first is the result integration mechnism to integrate results of all these base fuzzers together, and the second is the seed synchronization mechanism that makes those fuzzers cooperate with each other effectively.

1) Result Integration Mechanism: A straightforward solution to the ensemble architecture is the result integration mechanism, which has been adopted by most ensemble learning frameworks. Figure 2 presents the structure of result integration mechanism.

After the completion of the fuzzing session, we first collect results from all base fuzzers into a seeds pool. Then, we run these base fuzzers on the same target application for one time. The unique crashes, path coverage and branch coverage collected during the process are used to de-duplicate and triage the results. The result integration mechanism is easy to implement, and the result of integration outperforms any single result due to the diversity of base fuzzers.

However, the limitation of this mechanism is missing cooperation among these base fuzzers during their fuzzing processes — we just run these base fuzzers separately and integrate them in the end. An ensemble architecture only implementing this mechanism is insufficient. To improve the shallow ensemble fuzzer, a seed synchronization mechanism is proposed to cooperate with the result integration mechanism.

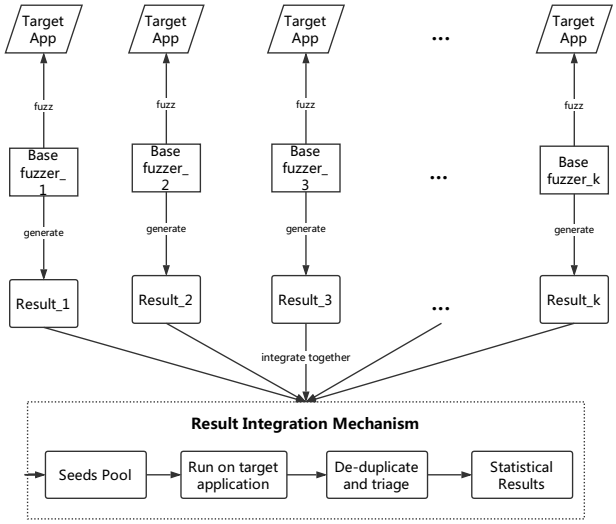


Fig. 2. The structure of result integration mechanism, which mainly contains the seed collection and result de-duplicate and triage.

2) Seed Synchronization Mechanism: The seed synchronization mechanism is presented in Figure 3. The main idea is to identify the interesting seeds (seeds that can execute new paths, cover new branchs or detect new crashs) of different base fuzzers and share those seeds during the fuzzing process.

When all base fuzzers have been established, a thread named `sync` will be created and is responsible for monitoring execution status of these fuzzing jobs. At the same time, an empty output directory `sync_dir` that is shared by all instances of base fuzzers will be created by the fuzzer initializer. Each fuzzer will keep its state in a separate subdirectory. If a hard-to-hit but interesting seed is generated by one fuzzer, the thread `sync` will put it into the shared directory `sync_dir`. Then, each base fuzzer will periodically rescan the top-level shared directory for any seed synchronized by the thread `sync`, and will corporate them into its own seed pool when they are deemed interesting. On account of this seed synchronization strategy, these base fuzzers can cooperate with each other efficiently when they are fuzzing the same target application.

The seed synchronization mechanism works together with the result integration mechanism to make these base fuzzers cooperate with each other deeply and effectively. At the completion of all fuzzing jobs, we use the result integration mechanism to help de-duplicate and triage these results of base fuzzers, and output a fuzzing report in the end. With the

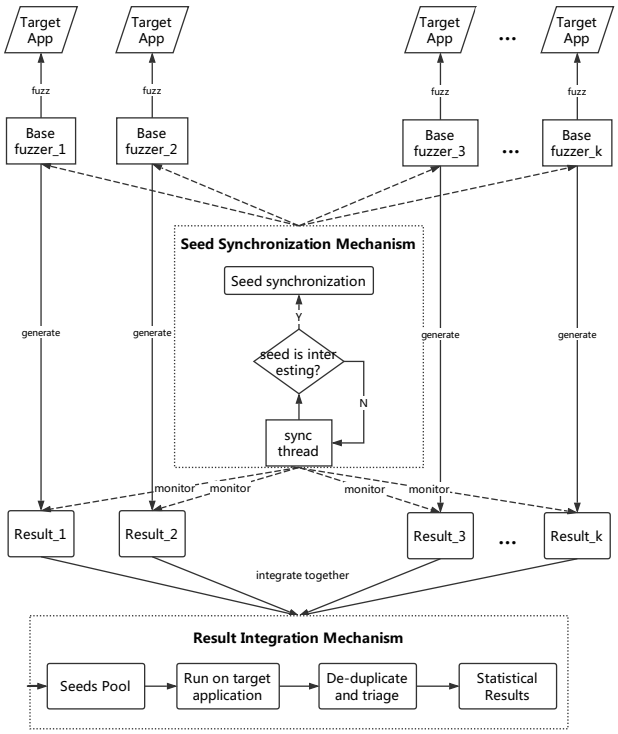


Fig. 3. The structure of seed synchronization mechanism, which mainly contains a sync thread for seed synchronization among base fuzzers.

help of these two core mechanisms, this ensemble architecture not only has a better generalization ability, but also improves the performance of base fuzzers effectively. There are many other effective and advanced ensemble methods in ensemble learning, such as Boosting, Bagging and Stacking, which could also be applied and customized basing on the architecture.

IV. Evaluation

A. Ensemble Fuzzer Implementation

We implement an ensemble fuzzing approach based on four stated-of-art fuzzers, including AFL, AFLFast, AFLGo and FairFuzz. They are chosen as the base fuzzers because:

- Easy integration. All the fuzzers are open-source and easy to use. It's easy to integrate those existing fuzzers into our ensemble architecture.
- Precise implementation. All the fuzzers have their core algorithms implemented already. We do not have to implement them on our own, which eliminates implementation errors and deviations introduced by us.
- Fair comparison. All the fuzzers preform very well and are the most recently and best fuzzers, and have premier results about performance comparisons with each other in literature. Without modifications, we can evaluate their generalization ability on real-world applications.

Two prototypes are developed: `EnFuzz⁻`, an ensemble fuzzer with the result integration mechanism; `EnFuzz`, an ensemble fuzzer with both mechanisms. The seed synchronization mechanism is implemented basing on the original AFL's parallel mode that each instance of AFL will periodically rescan the top-level `sync` directory for any interesting seeds. We monitor the states of each base fuzzer and synchronize their interesting seeds into the `sync` directory. The result integration mechanism is implemented basing on the AFL's benchmarking-only mode that executes initial seeds for one time. This mode can be used by setting the environment variable `AFL_BENCH_JUST_ONE = 1`. In AFL, the seed that exercises new paths, covers new branches or triggers new unique crashes will be collected. Therefore, we firstly combine all the seeds generated by these base fuzzers, then we feed them as the initial seeds to the AFL's benchmarking-only mode, and produce the final report.

B. Data and Environment Setup

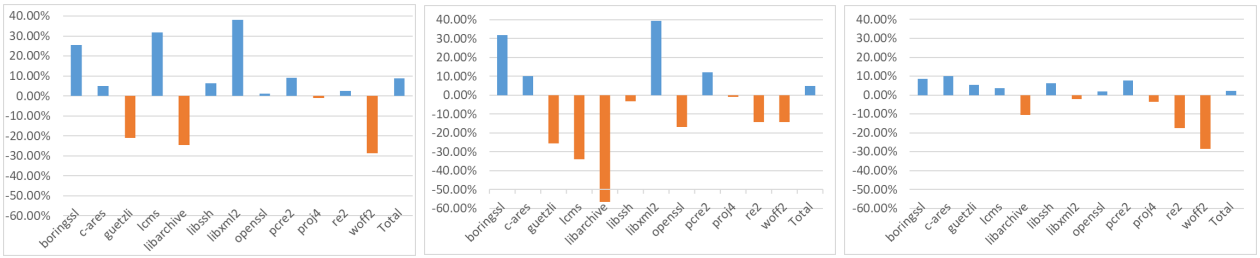
We evaluate our work basing on fuzzer-test-suite [5], a widely used third-party benchmark from Google. The test suite consists of popular open-source real-world applications, including well-known tools (e.g. libarchive, pcre2, woff2), image processing libraries (e.g. guetzli), communication protocol toolkits (e.g. openssl, boringssl, libssh), color management engines (e.g. lcms), regular expression engines (e.g. proj4, re2), and document processers (e.g. libxml2) etc. The benchmark is chosen to avoid the potential bias of the cases presented in literature, and its great diversity helps demonstrate the issue of generalization ability.

We use three widely used metrics to evaluate the results on these real-world applications, including the number of paths executed, branches covered and unique crashes triggered. The first two metrics can evaluate the coverage of the target applications, as coverage is one of the most important factors contributing to the success of fuzzers. In addition to coverage information, another important factor to fuzzers is the number of unique crashes detected. AFL distinguishes the crash by execution paths, therefore, some crashes could be caused by an identical root cause, while the others are not. But in general, the more crashes we found, the higher probability that more vulnerabilities could be identified. In other words, the number of unique crashes indicates the probability of finding vulnerabilities of target applications.

The experiment is conducted 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. Each binary is hardened by AddressSanitizer [8] to detect latent bugs. First we run each base fuzzer for 24 hours with one CPU core in single mode. Next, since both `EnFuzz⁻` and `EnFuzz` need at least four CPU cores to ensemble these four base fuzzers, we run each base fuzzer in parallel mode for 24 hours with four CPU cores, and this experiment setup ensures the computing resources usage of ensemble fuzzer is the same as any constituent base fuzzers in parallel mode.

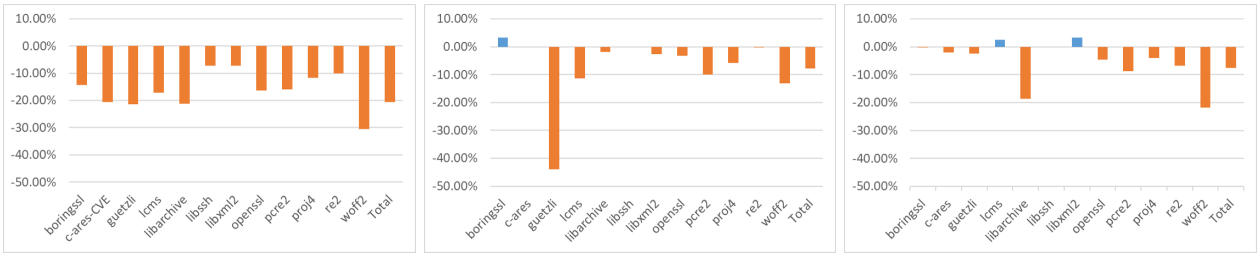
C. Generalization Ability Study

We choose AFL as the baseline, and compare other tools with AFL on path coverage to demonstrate the issue of the generalization ability first. Figure 4 shows the performance of paths executed by each base fuzzer compared with AFL in single mode. Considering that `EnFuzz⁻` and `EnFuzz` use



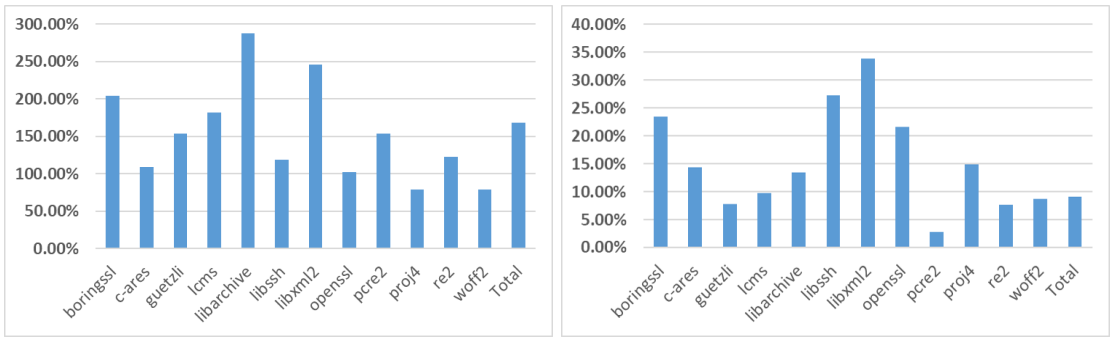
(a) performance of AFLFast in single thread (b) performance of FairFuzz in single thread (b) performance of AFLGo in single thread

Fig. 4. Paths covered by base fuzzers compared with AFL in single mode on a single core, blue means improvement and yellow means worse.



(a) performance of AFLFast in four threads (b) performance of FairFuzz in four threads (b) performance of AFLgo in four threads

Fig. 5. Paths covered by base fuzzers compared with AFL in parallel mode with four threads on four cores, blue means better and yellow means worse.



(b) compared with AFL in single thread (b) compared with AFL in four threads

Fig. 6. Paths covered by `EnFuzz` with four threads on four cores, compared with AFL in single mode on a single core and in parallel mode on four cores.

four times of computing resources compared with any constituent base fuzzer running in single mode, for fairness, we also collect the result of each base fuzzer running in parallel mode with four threads for 24 hours, and the result is presented in Figure 5. Figure 6 shows the performance of paths executed by `EnFuzz` compared with AFL in single mode and parallel mode respectively. From these results, we can get the following conclusions:

1) From both the results of Figure 4 and Figure 5, we find that compared with AFL, the three optimized fuzzers AFLFast, AFLGo and FairFuzz 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 applications. The performance of their fuzzing strategies is unstable, and cannot constantly perform well.
2) Comparing the result of Figure 4 with the result of Figure 5, we find that the performance of these base fuzzers in parallel mode are quite different from those in single mode. In single mode, the other three 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.
3) From the result of Figure 6, it reveals that `EnFuzz` always performs better than AFL on the target applications. Compared with AFL in single mode, `EnFuzz` covers 168% more paths in total, ranging from 70% to 280% for single case. For the same computing resources usage that AFL running in parallel mode with four CPU cores, `EnFuzz` covers 9.16% more paths than AFL, ranging from 3% to 34% for single case. The robustness and performance can be improved through ensemble approach.

It is reasonable to draw the conclusion that the generalization ability of each optimized fuzzer is greatly challenged by the complexity and diversity of those real-world applications, and it can be improved through the ensemble fuzzing approach. On one hand, the performance of those optimized strategies worked in single mode can not be directly scaled

to parallel mode, on the other hand, parallel mode is often used in industry practice. The ensemble approach can maintain the better performance in both modes, and more details are investigated and illustrated in the following subsections.

D. Evaluation in Detail

1) Intuitive evaluation in single mode: To evaluate the effectiveness of ensemble fuzzing, an intuitive way is to compare ensemble fuzzer with each constituent base fuzzer. Therefore, in this part, we present the detailed data of the following fuzzers: (1) base fuzzers, running in single mode, with one CPU core used; (2) `EnFuzz⁻`, featuring shallow ensemble, which only implements the result integration mechanism; (3) `EnFuzz`, featuring complete ensemble mechanism, which implements both result integration and seed synchronization mechanisms. Both `EnFuzz⁻` and `EnFuzz` run on four CPU cores, and each constituent base fuzzer run on one CPU core. Table I and Table II show the number of paths and branches covered by AFL, AFLFast, AFLgo, FairFuzz, `EnFuzz⁻` and `EnFuzz` for 24 hours. Table III shows the number of unique crashes detected by them.

TABLE I
NUMBER OF PATHS FOR INTUITIVE SINGLE MODE

Project	AFL	AFLFast	FairFuzz	AFLGo	EnFuzz⁻	EnFuzz
boringssl	1334	1674	1760	1448	2590	4058
c-ares	80	84	88	88	149	167
guetzli	1382	1090	1030	1456	2066	3501
lcms	656	864	434	680	1056	1846
libarchive	3756	2834	1630	3356	4823	14563
libssh	64	68	62	68	109	140
libxml2	5762	7956	8028	5642	11412	19928
openssl	2456	2482	2040	2500	3949	4976
pcre2	32310	35288	36176	34756	57721	81830
proj4	220	218	218	212	362	393
re2	5860	6014	5016	4836	9053	13019
woff2	14	10	12	10	19	25
Total	53894	58582	56494	55052	93309	144446

TABLE II
NUMBER OF BRANCHES FOR INTUITIVE SINGLE MODE

Project	AFL	AFLFast	FairFuzz	AFLGo	EnFuzz⁻	EnFuzz
boringssl	2645	3054	3115	2817	3210	3996
c-ares	126	122	126	126	285	285
guetzli	1913	1491	1428	1981	2074	3316
lcms	2216	2755	935	2248	2872	4054
libarchive	4906	3961	2387	4565	6092	12689
libssh	604	604	604	604	613	614
libxml2	10082	12407	12655	10095	14428	17657
openssl	3978	4015	3883	4017	4037	4176
pcre2	27308	29324	28404	28441	32471	51801
proj4	264	260	260	262	267	267
re2	15892	15970	15073	14717	16300	18070
woff2	114	112	114	112	120	120
Total	70048	74075	68984	69985	82769	117045

First four columns of Table I show the performance of the base fuzzers. The original AFL performs the best on 3 applications, but performs the worst only on 3 applications. However, according to literature reviews, those optimizations should outperform AFL. More specifically, AFLGo performs the best only on 4 applications, and the worst on 4 other applications. AFLFast just performs the best on 3 applications, and performs the worst on 1 application. FairFuzz also performs the best only on 4 applications, but worst on the other 5 applications. Although the total number of paths covered improves slightly, the deviation for each application is huge, ranging from -57% to 38%, as demonstrated in Figure 4.

TABLE III
NUMBER OF UNIQUE CRASHES FOR INTUITIVE SINGLE MODE

Project	AFL	AFLFast	FairFuzz	AFLGo	EnFuzz⁻	EnFuzz
boringssl	0	0	0	0	0	0
c-ares	5	3	4	5	12	18
guetzli	0	0	0	0	0	0
lcms	0	0	0	0	0	7
libarchive	0	0	0	0	0	27
libssh	0	0	0	0	0	4
libxml2	0	7	0	0	7	13
openssl	10	7	0	11	21	86
pcre2	354	573	489	314	702	4234
proj4	0	0	0	0	0	15
re2	0	0	0	0	0	5
woff2	0	0	0	0	0	3
Total	369	590	493	330	742	4412

From the first four columns in Table II and Table III, we can get the same observation that the performance of these optimized fuzzers vary widely on different applications. Although the total number of covered branches and unique crashes improved slightly, the deviation of each application is huge. Take FairFuzz for example, on libxml2, the rare branch fuzzing strategy guides FairFuzz into deeper areas and covered more branches. However, on libarchive, this strategy fails. FairFuzz spends too much time in deep areas, ignoring breadth search. Unlike libxml2, the breadth first search strategy of other fuzzers is more effective on libarchive.

The last two columns of Table I, II and III, show the performance of ensemble fuzzing. For `EnFuzz⁻`, the shallow ensemble fuzzer only implements the result integration mechanism. It always performs better than the four base fuzzers on the applications in experiment, in terms of paths executed, branches covered and unique crashes detected, according to the sixth columns of the tables separately.

In detail, compared with AFL, AFLFast, AFLGo and FairFuzz, `EnFuzz⁻` achieves 73.1%, 59.3%, 65.2% and 69.5% more paths executed, and 18.2%, 11.7%, 20.1% and 18.3% more branches covered. Benefit from the increased coverage, `EnFuzz⁻` finds 101%, 25.8%, 50.5% and 24.8% more unique crashes respectively.

For `EnFuzz` that implements both seed synchronization mechanism and result integration mechanism, the seventh column shows its evident advantage over `EnFuzz⁻`. In detail, compared with AFL, AFLFast, AFLGo and FairFuzz, `EnFuzz` executes 171%, 150%, 159% and 166% more paths, covers 67.1%, 58.0%, 70.1% and 67.2% more branches and triggers 1095%, 647%, 795% and 1237% more unique crashes respectively. Compared with EnFuzz⁻, `EnFuzz` executes 54.8% more paths, covers 41.4% more branches and triggers 494.6% more crashes. We can conclude that the seed synchronization mechanism is effective and greatly improves the performance of ensemble fuzzers.

From the description above, we can draw the conclusion

that ensemble fuzzing improves the generalization ability of individual fuzzers greatly. However, each base fuzzer is running in single mode with one CPU core, and the ensemble fuzzer needs four CPU cores at least — one CPU core for each constituent base fuzzer. The improvements in performance presented in these three tables come at the cost of more computing resources usage.

For a fairer comparison, we need to allocate four CPU cores for each base fuzzer in two different ways: (1) For each base fuzzer, we run four instances of it, then we apply result integration mechanism to help de-duplicate and triage the results. In this way the computing resources usage is same, and we find the results are almost the same as in these three tables because there is little diversity between these four instances of the same fuzzer. (2) We run each base fuzzer in parallel mode with four CPU cores used, and then we also apply result integration mechanism to help de-duplicate and triage the results. The final results not only demonstrate the improvements of ensemble fuzzing, but also reveal some other interesting points. More details will be introduced in the advanced evaluation in parallel mode below.

2) Advanced evaluation in parallel mode: In the intuitive evaluation in single mode, both `EnFuzz⁻` and `EnFuzz` use 4 times computing resources compared with base fuzzers. To eliminate the unfairness, we need to allocate the same computing resources to these base fuzzers. Luckily, most fuzzers support a parallel mode which allows using multiple CPU cores to fuzz one target application in parallel. And for industry practice, fuzzers generally work in parallel mode. Therefore, we run each base fuzzer in parallel mode with four CPU cores used for 24 hours respectively, in this way, the computing resources usage of these base fuzzers is the same as `EnFuzz⁻` and `EnFuzz`. The results are presented in Table IV, V and VI, which show the number of paths executed, branches covered and unique crashes detected by AFL, AFLFast, AFLGo, FairFuzz, `EnFuzz⁻` and `EnFuzz`.

The first four columns of Table IV still reveal the issue of the generalization ability in those base fuzzers, as they perform variously on different applications in parallel mode. AFL performs better than the other 3 base fuzzers on 9 applications, FairFuzz performs the best on the other 4 applications, AFLGo only performs the best on lcms and libssh, and AFLFast does not perform the best on any application. Table V and Table VI show similar results on branch coverage and unique crashes.

Table IV and Table I show that the performance of base fuzzers in parallel mode is quite different from that in single mode. Compared with other optimized fuzzers, original AFL performs the best on 9 applications in parallel mode with 4 CPU cores used. While in single mode with one CPU core used, the original AFL only performs the best on 3 applications. For the total number of paths executed, AFL performs the best and AFLFast performs the worst in parallel mode. In single mode, the situation is exactly the opposite.

Our evaluation demonstrates that most optimized fuzzing strategies could be useful and work well in single mode, but fail in parallel mode. In fact, according to the experiments, compared with AFLFast, AFLGo and FairFuzz, AFL generally performs better than the other three base fuzzers in parallel mode. 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.

As for the performance of ensemble fuzzing versus those base fuzzers running in parallel mode, the last two columns of Table IV, V and VI are the same with Table I, II and III, respectively. Compared with AFL, AFLFast, AFLGo and FairFuzz in parallel mode with four CPU cores used, the performance of the shallow ensemble fuzzing `EnFuzz⁻` is the worst. It only covers 70.5%, 88.7%, 76.4% and 76.3% paths, exercises 74.9%, 79.2%, 79.5% and 78.6% branches, and discovers 21.3%, 36.5%, 23.3% and 23.5% unique crashes, respectively.

TABLE IV
NUMBER OF PATHS FOR ADVANCED PARALLEL MODE

Project	AFL with 4 threads	AFLFast with 4 threads	FairFuzz with 4 threads	AFLGo with 4 threads	EnFuzz⁻	EnFuzz
boringssl	3286	2816	3393	3275	2590	4058
c-ares	146	116	146	143	149	167
guetzli	3248	2550	1818	3169	2066	3501
lcms	1682	1393	1491	1725	1056	1846
libarchive	12842	10111	12594	10449	4823	14563
libssh	110	102	110	110	109	140
libxml2	14888	13804	14498	15373	11412	19928
openssl	4090	3425	3956	3901	3949	4976
pcre2	79581	66894	71671	72587	57721	81830
proj4	342	302	322	328	362	393
re2	12093	10863	12085	11284	9053	13019
woff2	23	16	20	18	19	25
Total	132331	105138	122104	122362	93309	144446

TABLE V
NUMBER OF BRANCHES FOR ADVANCED PARALLEL MODE

Project	AFL with 4 threads	AFLFast with 4 threads	FairFuzz with 4 threads	AFLGo with 4 threads	EnFuzz⁻	EnFuzz
boringssl	3834	3635	3894	3634	3210	3996
c-ares	285	276	285	285	285	285
guetzli	3022	2723	1514	2927	2074	3316
lcms	3985	3681	3642	3891	2872	4054
libarchive	10580	9267	8646	10144	6092	12689
libssh	614	614	614	614	613	614
libxml2	15204	14845	14298	14578	14428	17657
openssl	4079	4004	4021	4026	4037	4176
pcre2	50558	48004	49430	47624	32471	51801
proj4	267	267	267	267	267	267
re2	17918	17069	17360	17136	16300	18070
woff2	120	120	120	120	120	120
Total	110466	104505	104091	105246	82769	117045

This shallow ensemble method is inefficient because we only implement result integration mechanism that combines these base fuzzers in the final result, thus there is no cooperation among these base fuzzers during their fuzzing processes. In contrast, for those base fuzzers running in parallel mode with four CPU cores, the four instances of base fuzzers cooperate with each other by sharing their interesting seeds,

TABLE VI
NUMBER OF UNIQUE CRASHES FOR ADVANCED PARALLEL MODE

Project	AFL with 4 threads	AFLFast with 4 threads	FairFuzz with 4 threads	AFLGo with 4 threads	EnFuzz$^-$	EnFuzz
boringssl	0	0	0	0	0	0
c-ares	15	12	14	15	12	18
guetzli	0	0	0	0	0	0
lcms	6	5	6	6	0	7
libarchive	0	0	0	0	0	27
libssh	0	0	0	0	0	4
libxml2	11	7	9	10	7	13
openssl	68	8	66	54	21	86
pcre2	3360	2001	3078	3072	702	4234
proj4	14	0	4	6	0	15
re2	3	0	1	0	0	5
woff2	2	0	0	0	0	3
Total	3479	2033	3178	3163	742	4412

which helps improve the performance of these base fuzzers greatly and is even better than EnFuzz$^-$.

For the complete ensemble fuzzer EnFuzz which implements both the result integration and the seed synchronization mechanisms, compared with AFL, AFLFast, AFLGo and FairFuzz running in parallel mode with four CPU cores used, it always executes more paths and covers more branches on all applications. In total, it covers 9.16%, 39.2%, 19.9% and 20.0% more paths and achieves 5.96%, 12.0%, 21.4% and 11.1% more branches covered respectively. Benefit from the increased coverage, EnFuzz triggers 26.8%, 117%, 38.8% and 39.5% more unique crashes in total. If we run those base fuzzers and EnFuzz with more CPU cores, e.g. 8 or 12, the performance improvements remain.

There are two main reasons for the high performance of EnFuzz. The first is the effective seed synchronization mechanism. With the help of this strategy, all interesting but hard-to-hit seeds are synchronized to all base fuzzers, enabling effective communication and corporation among base fuzzers. The second is the diversity of base fuzzers. Fuzzers with different fuzzing strategies cover different paths and branches. When we combine them together, the ensemble results are better than any of the constituent base fuzzers.

In addition, we notice that there are some deviation of improvements among different applications, ranging from 3% to 34%. The main reason is the code sizes of target applications. We observe that the applications with low performance improvements are c-ares, libssh, and proj4. We find the code sizes of these applications are relatively small, which means most branches of these application have been covered in single mode or parallel mode by each base fuzzer alone. There is little space for improvement, even for an ensemble fuzzer. The larger the code size is, the higher the improvement in performance of ensemble fuzzing is.

From these comparisons and statistics, we can conclude that (1) the base fuzzers in parallel mode perform much better than those in single mode; (2) the issue of generalization ability exists in both single mode and parallel mode; (3) the optimization of fuzzing strategy works in single mode on certain situations can not be applied to parallel mode directly, and in most cases the performance of the optimization is worse in parallel mode; (4) ensemble fuzzer can help mitigate the lack of generalization ability and improve the performance of base fuzzers in both single mode and parallel mode.

V. DISCUSSION

Basing on Google's fuzzer-test-suite, we demonstrate this ensemble fuzzing approach outperforms any other base fuzzers on all applications. However, some limitations still threaten the performance of ensemble fuzzing. The representative limitations and the workarounds are discussed below.

The first potential threat is the insufficient diversity of the base fuzzers. In section III-A for the base fuzzers selection, we propose three different dimensions, including the coverage information granularity diversity, seed generation strategy diversity, and seed mutation as well as selection strategy diversity. For the implementation of EnFuzz, we select AFL, AFLFast, AFLGo and FairFuzz as the base fuzzers. They are open-source and can demonstrate the issue of generalization ability easily based on their previous evaluations and comparisons, but they partially cover those dimensions and may be insufficient. AFLFast aims at accelerating fuzzing, AFLGo aims at target-place directed fuzzing, and FairFuzz aims at rare branch fuzzing. We can integrate other available fuzzers to improve the diversity of base fuzzers. Another problem related is regarding those three dimensions equally. For example, compared with same-type fuzzers' strategies (e.g. mutation or selection strategy), the diversity of different types of fuzzers (e.g. generation-based and mutation-based) is usually larger. Therefore, difference in fuzzers' types contribute more to the improvement in performance. We can assign different weights to different dimensions after empirical studies. Also, based on the three basic dimensions, we can try to adapt some diversity measures in ensemble learning to ensemble fuzzing, such as sub-sampling the training examples.

The second potential threat is the mechanism scalability of the ensemble architecture. In section III-B about the ensemble architecture design, we propose the seed synchronization mechanism and result integration mechanism. EnFuzz$^-$ only focuses on integrating the fuzzing results of base fuzzers, and its limited performance can not be improved greatly. Inspired from parallel mode of many fuzzers, EnFuzz implements the seed synchronization mechanism to cooperate each base fuzzer together effectively, and the performance is much improved and is better than any of the constituent base fuzzers with same computing resources usage. However, these two ensemble mechanisms can still be improved for better scalability on different applications and fuzzing tasks. Take seed synchronization mechanism for example, EnFuzz only synchronize the coarse-grained information — interesting seeds, rather than the fine-grained information. For example, we could synchronize execution trace of each base fuzzers to improve their effectiveness in cooperation.

We can also apply some effective ensemble mechanisms in ensemble learning to ensemble fuzzing, such as Boosting, Bagging and Stacking. For example, Boosting is a widely used ensemble mechanism which will reweigh the base learner dynamically to improve the performance of ensemble learner. To

implement this idea in ensemble fuzzing, we could start up a *sync* thread to monitor the execution status of all base fuzzers, and reassign the computing resources usage accordingly.

VI. Conclusion

In this paper, we first propose ensemble fuzzing for software vulnerability detection. It ensembles multiple fuzzing strategies, solves the issue of generalization ability in different base fuzzers, and obtains better performance than that of any constituent base fuzzer alone. To apply the idea of ensemble learning to ensemble fuzzing, we bridge two gaps. First, we define the diversity of base fuzzers and propose the method to select base fuzzers — as diverse as possible. Then, we propose a concrete ensemble architecture with two effective ensemble mechanisms — seed synchronization and result integration mechanisms. For evaluation, we implement `EnFuzz`, a prototype of ensemble fuzzing. We evaluate `EnFuzz` with base fuzzers on a third-party benchmark, Google's fuzzer-test-suite, which consists of real-world applications. `EnFuzz` always outperforms other popular base fuzzers in terms of unique crashes, path and branch coverage in both single mode and parallel mode. We also find that most of the existing optimizations for fuzzing strategies work well in single mode, but fail in parallel mode. Our implemented ensemble architecture can be utilized to integrate other base fuzzers for industry practice easily. Our future work will focus on two directions: the first is to try some other dimensions of diversity in base fuzzers; the second is to improve the ensemble architecture for better generalization ability.

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

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

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

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

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

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

[9] Eric Bauer and Ron Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine learning*, 36(1-2):105–139, 1999.

[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS17)*, 2017.

[11] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043. ACM, 2016.

[12] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[13] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 725–741. IEEE, 2015.

[14] Thomas G Dietterich. Machine-learning research. *AI magazine*, 18(4):97, 1997.

[15] Thomas G Dietterich. An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine learning*, 40(2):139–157, 2000.

[16] Thomas G Dietterich. Ensemble learning. *The handbook of brain theory and neural networks*, 2:110–125, 2002.

[17] Michael Eddington. Peach fuzzing platform. *Peach Fuzzer*, page 34, 2011.

[18] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

[19] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *ACM Sigplan Notices*, volume 43, pages 206–215. ACM, 2008.

[20] Lars Kai Hansen and Peter Salamon. Neural network ensembles. *IEEE transactions on pattern analysis and machine intelligence*, 12(10):993–1001, 1990.

[21] Sam Hocevar. zzuf - multi-purpose fuzzer. http://caca.zoy.org/wiki/zzuf, 2007. [Online; accessed 10-April-2018].

[22] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *USENIX Security Symposium*, pages 445–458, 2012.

[23] Anders Krogh and Jesper Vedelsby. Neural network ensembles, cross validation, and active learning. In *Advances in neural information processing systems*, pages 231–238, 1995.

[24] Ludmila I Kuncheva and Christopher J Whitaker. Measures of diversity in classifier ensembles and their relationship with the ensemble accuracy. *Machine learning*, 51(2):181–207, 2003.

[25] Caroline Lemieux and Koushik Sen. Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage. *arXiv preprint arXiv:1709.07101*, 2017.

[26] David W Opitz and Richard Maclin. Popular ensemble methods: An empirical study. *J. Artif. Intell. Res.(JAIR)*, 11:169–198, 1999.

[27] Robert E Schapire. The strength of weak learnability. *Machine learning*, 5(2):197–227, 1990.

[28] Emin Gün Sirer and Brian N Bershad. Using production grammars in software testing. In *ACM SIGPLAN Notices*, volume 35, pages 1–13. ACM, 1999.

[29] Vladimir Svetnik, Andy Liaw, Christopher Tong, J Christopher Culberson, Robert P Sheridan, and Bradley P Feuston. Random forest: a classification and regression tool for compound classification and qsar modeling. *Journal of chemical information and computer sciences*, 43(6):1947–1958, 2003.

[30] Kai Ming Ting and Ian H Witten. Issues in stacked generalization. *J. Artif. Intell. Res.(JAIR)*, 10:271–289, 1999.

[31] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security*, pages 581–601. Springer, 2016.

[32] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing, 2017.

[33] Mingzhe Wang, Liang Jie, Yuanliang Chen, Yu Jiang, Jiao Xun, Liu Han, Zhao Xibin, and Sun Jiaguang. Safl: Increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Software Engineering Companion (ICSE-C), 2018 IEEE/ACM 40th International Conference on Software Engineering*. IEEE, 2018.

[34] David H Wolpert. Stacked generalization. *Neural networks*, 5(2):241–259, 1992.

[35] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 2011.

[36] Michal Zalewski. American fuzzy lop, 2015.

[37] Zhi-Hua Zhou. Ensemble learning. *Encyclopedia of biometrics*, pages 411–416, 2015.