Industrial Oriented Evaluation of Fuzzing Techniques

Mingzhe Wang
School of Software
Tsinghua University, Beijing, China

Jie Liang
School of Software
Tsinghua University, Beijing, China

Chijin Zhou
School of Software
Tsinghua University, Beijing, China

Yuanliang Chen
School of Software
Tsinghua University, Beijing, China

Zhiyong Wu
School of Software
Tsinghua University, Beijing, China

Yu Jiang✉
School of Software (Correspondence Author)
Tsinghua University, Beijing, China

***Abstract*—Fuzzing is a promising method for discovering vulnerabilities. Recently, various techniques are developed to improve the efficiency of fuzzing, and impressive gains are observed in evaluation results. However, evaluation is complex, as many factors affect the results, for example, test suites, baseline and metrics. Even more, most experiment setups are lab-oriented, lacking industrial settings such as large code-base and parallel runs. The correlation between the academic evaluation results and the bug-finding ability in real industrial settings has not been sufficiently studied.**

In this paper, we test representative fuzzing techniques to reveal their efficiency in industrial settings. First, we apply typical fuzzers on academic widely used small projects from LAVA-M suite. We also apply the same fuzzers on large practical projects from Google's fuzzer-test-suite, which is rarely used in academic settings. Both experiments are performed in both single and parallel run. By analyzing the results, we found that most optimizations working well on LAVA-M suite fail to achieve satisfying results on Google's fuzzer-test-suite (e.g. compared to AFL, QSYM detects 82x more synthesized bugs in LAVA-M, but only detects 26% real bugs in Google's fuzzer-test-suite), and the original AFL even outperforms most academic optimization variants in industry widely used parallel runs (e.g. AFL covers 13% more paths than AFLFast). Then, we summarize common pitfalls of those optimizations, analyze the corresponding root causes, and propose potential directions such as orchestrations and synchronization to overcome the problems. For example, when running in parallel on those large practical projects, the proposed horizontal orchestration could cover 36%-82% more paths, and discover 46%-150% more unique crashes or bugs, compared to fuzzers such as AFL, FairFuzz and QSYM.

I. Introduction

Since 1990s, researchers has proposed numerous techniques to improve the effectiveness of fuzzers [24]. It begins with blackbox fuzzers, where the fuzzer has no knowledge about the internals of the program. The basic idea is simple: generate a large number of random inputs for the program, and then catch abnormal behavior which indicates bugs. Following works accelerate blackbox fuzzing by specifying the input structure.

A prominent improvement of blackbox fuzzing is greybox fuzzing, which guides fuzzing by coverage information. A representative work of greybox fuzzers is American Fuzzy Lop (AFL) [33]. Its genetic algorithm is extensively studied, and many works [3, 20, 23, 27, 34, 21] are derived from it. By analyzing programs in source code or binary form, whitebox fuzzers have more-detailed knowledge compared to greybox fuzzers. Symbolic execution-based whitebox fuzzers [4, 2] systematically explore the state space of a program by automatically construct inputs exercising a predefined path. Taint analysis-based whitebox fuzzers [26] infer bytes that impact more and limit mutation to those bytes accordingly. The approach of hybrid fuzzing is combing fuzzers together to play to their strengths. For example, some works [30, 31, 32] combine greybox fuzzer and whitebox fuzzer, while some works [8, 22] combine different types of greybox fuzzers.

Creative fuzzing techniques are flourishing, so are evaluation methods inside the papers. Take runtime environment for example, SYMFUZZ [5] runs fuzzers for an hour, while Orthrus [29] extends the timeout to a week. As for test suite, some works focus on real-world programs [14], while some works are evaluated on synthesized bugs [10]. Metrics used to compare against each other are even more problematic, as the same metric has different implementations. For instance, AFL treats crashes with different coverage pattern as "unique crashes"; VUzzer [26] deduplicates crashes considering crash location, memory operation and signals triggered [17]; Angora [6] even does not supply the standard for crash triage.

Most optimizations work well and outperforms their comparative targets in their evaluation and experiment settings. But the diversity mentioned above poses a threat to evaluation, because different evaluation setup harms comparability, and improper evaluation harms authenticity. Recently, Klees et al. [19] took the first tentative steps towards better evaluation and provides some guidelines. However, the academia has not reached a consensus on a concrete procedure for evaluation. Besides, there is no result to demonstrate the correlation between the academic results and the bug finding ability in real industrial settings.

In this paper, in order to reveal the efficiency of existing academic fuzzing optimizations in industrial settings, we systematically evaluate typical fuzzers on practical large projects with parallel runs. Results on academic widely used small projects from LAVA-M with single run are collected for further comparison, and the metrics selected are bug count, branch

count and path count.

We find that most optimizations working well in academic settings fail to achieve satisfying results in industrial settings. We analyze the common pitfalls affecting the effectiveness of a fuzzing technique in industrial settings, investigate the root causes behind the pitfalls, and propose initial solutions to the problems. We observe that academic fuzzers have unstable performance in different projects (e.g. when considering path coverage, FairFuzz performs better than AFL in 14 programs, but fails to keep up the advantage in the other 14 programs), and propose horizontal and vertical orchestration of different fuzzers as a solution. We observe that fuzzers fail to keep up the performance improvements when running in parallel (e.g. when comparing AFL and AFLFast on total path coverage in Google's fuzzer-test-suite, the gain of AFLFast is 12% in single mode, but -11% in parrallel mode), and propose synchronization of statistics as a solution. We observe that symbolic execution has a bottleneck on large-scale programs (e.g. compared to AFL, QSYM detects 82x more synthesized bugs in LAVA-M, but only detects 26% real bugs in Google's fuzzer-test-suite), and propose lightweight symbolic execution as a solution. Our preliminary experiments indicate that the potential solutions are effective, and may provide directions for future research. For example, compared to fuzzers such as AFL, FairFuzz and QSYM, the proposed horizontal orchestration could cover 36%-82% more paths, and discover 46%-150% more unique crashes or bugs, when running in parallel on large practical projects.

The rest of the paper is organized as follows: Section II investigates existing fuzzing works and briefly describes the techniques used in the works. Section III summarizes evaluation methods commonly used in the fuzzing works. Section IV describes the experiment setup following industrial settings. Section VI explores the possible solutions to the problems reflected by the results. Section VII draws the conclusion.

II. Fuzzing Techniques

In this section, we systematically investigate typical fuzzing works, and summarize their techniques in Table I.

TABLE I
List of Representative Fuzzing Works

Fuzzer	Origin	Type	Characteristics
fuzz	CACM'90	Blackbox	First fuzzer
zzuf	Industry'07	Blackbox	Perturbing API calls
BFF	Industry'10	Blackbox	Tuning via machine learning
Radamsa	Industry'07	Blackbox	Heuristics-based mutation
AFL	Industry'13	Greybox	Coverage-guided fuzzing
AFLFast	CCS'16	Greybox	Frequency-aware scheduling
FairFuzz	ASE'18	Greybox	Structure-aware mutation
libFuzzer	Industry'13	Greybox	In-process fuzzing
SAGE	NDSS'08	Whitebox	Whitebox fuzzing
KLEE	OSDI'08	Whitebox	Coverage-guided path selection
Angora	S&P'18	Whitebox	Search with gradient descent
Driller	NDSS'16	Hybrid	Selective symbolic execution
QSYM	Security'18	Hybrid	Fuzzing-oriented symbolic execution

A. Blackbox Fuzzing

The term "fuzz" was first proposed in 1990 [24]. Designed for testing UNIX utilities, this work consists of three components: a script to repeat the test, program `fuzz` to generate random characters, and program *ptygig* to help execute the test. The shell script first invokes `fuzz` to generate the random output, and redirects it to the UNIX utility being tested, and then waits for crash, hang or success. Interactive programs can not be tested in such way, as they read commands from terminal. As a workaround, this work provides utility `ptygig`, which passes the random characters to the target program via pseudo-terminal. The procedure above is considered as *blackbox fuzzing*, because the fuzzer has no detailed knowledge of the program being tested.

Following works mainly focus on the input generation stage and execution stage. **zzuf** [16] intercepts file reading related functions such as `open` and `fread` to injects randomness into the results. This mechanism is more effective than generating a completely new file, which differs too much to be accepted by the program. zzuf also intercepts `malloc` to check for heap-related memory corruptions. CERT Basic Fuzzing Framework (**BFF**) [11] enhances blackbox fuzzing with online machine learning. BFF organizes blackbox fuzzing into iterations. Each iteration has a limited program execution count, and zzuf is invoked for actual testing. Between each iteration, the parameters of zzuf are optimized with the Multi-Armed Bandit model. The parameters include which seed files are used for mutation, and the proportion of a seed for mutation. BFF is more than a blackbox fuzzer. It complements the post-fuzzing analysis with a set of utilities, such as test-case minimization. These analyses are not only a complement to blackbox fuzzers, but also important components in greybox fuzzers. **Radamsa** [15] leverages heuristics to generate inputs more efficiently. Not limited to byte-level mutations where most fuzzers does, Radamsa mutates with a set of high-level mutate operands, e.g. line swaps, malformed Unicode codepoints and mishandled ASCII strings.

B. Greybox Fuzzing

Greybox fuzzers guide fuzzing with coverage information to accelerate fuzzing, because the prerequisite of triggering a bug is covering the buggy code. Zalewski developed the first greybox fuzzer, i.e. **AFL** [33], in 2013. AFL first instruments the target program. At the beginning of each basic block, the injected code increases a counter representing current basic block transition (branch). After the execution of an input, AFL analyzes all the counters to detect the change in branch coverage. With the feedback counters available, AFL is able to determine the usefulness of a newly generated input, and prioritize the most promising seeds accordingly. Generally speaking, greybox fuzzers first select seeds with evolutionary algorithms, and then mutate a seed with various operands, and finally run the test with the new input.

Possibilities of optimization exist in each stage. **AFLFast** [3] optimizes the selection stage. By combining multiple per-input feedback, AFLFast constructs a novel global view of

each branch. More specifically, AFLFast favors the seed exercising the least-chosen branch. The strategy is implemented in power schedule, where the time mutating a chosen seed is adjusted basing on the rarity. **FairFuzz** [20] optimizes the mutation stage. The core idea is to preserve a seed exercising rare branch when mutating. To do so, FairFuzz experimentally mutate each byte of the seed, just to detect whether mutating the byte results in divergence from the rare branch. Additionally, FairFuzz also proposes an alternative selection algorithm, which keeps mutating the seed exercising most rare branch. **libFuzzer** [25] optimizes the execution part. As an in-process fuzzer, libFuzzer removes the overhead of process-based fuzzer, for instance spawning a new process and writing the input to file. libFuzzer requires the user to define a function accepting the pointer to input buffer and the length of input. The function is repeatedly called to execute inputs.

C. Whitebox Fuzzing

Whitebox fuzzers is able to systematically explore the state space of the target program, because whitebox fuzzers have access to more detailed information compared to greybox fuzzers. The first whitebox fuzzer, **SAGE** [13], runs the target program with a concrete input and collects the trace. The trace is then used to collect constraints affecting branches on the path. Aiming to generate a new path, SAGE systematically negates the constraints, and generates a new concrete input via constraint solver. The trace with concrete value helps symbolic execution by simplifying constraints and modeling library calls.

Instead of analyzing a concrete trace *offline* like what SAGE does, **KLEE** [4] explores the state space *online*. KLEE collects the constraints of the target program from the entry point. When a branch is reached, KLEE consults the constraint solver to determine if it is possible to follow each destination. If both are possible, KLEE forks to trace each path separately. When an instruction may lead to error (such as memory access), KLEE checks if the error can be triggered, and generates concrete value if so. The online approach forks on branches, which may exhaust the memory; however, it is free of repeated collection of the constraints close to the entry point. Akin to SAGE, **Angora** [6] aims to flip a specific branch to improve coverage. Nevertheless, Angora does not work at path level with symbolic execution. It models a branch as function $f(\mathbf{x})$, where $\mathbf{x}$ denotes the bytes affecting the branch. To build the correlation between the branch and bytes in input affecting the branch, Angora first performs dynamic taint analysis. The result is further processed to infer the type behind the bytes. Finally the function is optimized with gradient descent by only mutating the "hot bytes".

D. Hybrid Fuzzing

Hybrid fuzzing is a technique to magnify the strengths of fuzzing and symbolic execution. In theory, whitebox fuzzing is able to explore all states including branches. However, constraint collection and solver calls slow down symbolic execution by several orders of magnitude. Symbolic execution is precise but slow, and fuzzing is fast but imprecise. **Driller** [30] first launches its fuzzing engine to explore the program. When fuzzing gets "stuck", Driller switches to concolic execution. As the statistics of the fuzzing stage, the concolic execution engine generates a constraint leading the path to an unexplored area. Then the constraint is solved and the concrete value is passed to the fuzzing engine to repeat the process. **QSYM** [32] observes that performance of symbolic execution is the main factor limiting hybrid fuzzing. First, QSYM directly emulates at instruction level to remove the overhead of intermediate representations in constraint collection phase. Second, QSYM removes ineffective snapshot, which is not applicable for assisting fuzzing. Finally, QSYM optimistically solves parts of the constraints, as fuzzing is able to bypass some checks.

III. Typical Academic Evaluation

In this section, we review programs and metrics typically used by the academia for evaluating and comparing fuzzers.

A. Test Suites

TABLE II
List of Academic Test Suites

Test Suite	Stable	Known Bug	Real Bug	Real Program
Utilities	No	No	Yes	Yes
LAVA-M	Yes	Yes	No	Yes
CGC	Yes	Yes	Yes	No

Test suite used in evaluation begins with **UNIX utilities** [24]. The utilities are simple in I/O, but complex in logic. While some works [4] still evaluate on it, an alternative option is to fuzz **file-related utilities**, because files in complex formats are difficult to parse correctly. Take portable document format (PDF) for example, ISO 32000-2:2017 [9], the document specifying the format of PDF, has a total number of 971 pages. Programs accepting document, image, video, executable or network capture as inputs are popular targets for fuzzing tasks.

To quantitatively measure whether a bug can be detected by a fuzzer, the prerequisite is knowing the ground truth. Utilities are not suitable for composing a test suite — when the maintainers release a new version, vulnerabilities are fixed and new logic is added. There also misses a well-recognized standard for choosing utilities, granting that the versioning problem is resolved. Besides, test suites with up-to-date utilities are unknown in the existence of a bug, much less the number of total bugs.

To construct a stable ground truth, one approach is synthesizing the test suite by automatically injecting bugs into the program, i.e. artificial bugs from real programs. **LAVA-M** [10] is a test suite constructed by injecting out-of-bound reads and writes into GNU coreutils. Another approach is intentionally write programs with bugs and construct a test suite with such programs, i.e. real bugs from artificial programs. If correctly constructed, bugs injected in this approach are more authentic. **Cyber Grand Challenge** [12] follows this path,

and 131 programs with vulnerabilities are written by security researchers. However, to simplify the interaction between the program and the OS, only 7 specially crafted syscalls are allowed to use within the program.

In conclusion, a good test suite should contain real-world programs with fixed source code, and the bugs inside those programs should be authentic and already known. However, as far as we know, no test suites that academia prefers to evaluate fuzzers on fulfill all the properties.

B. Evaluation Metrics

Even on the very same experiment with identical test suite, different conclusions may be drawn if metrics used are different. Since the potential points of optimization broadly exist in fuzzing, different works focus on different points, and the metrics used for evaluation differ consequently. However, the ultimate standard for fuzzers remains the same: the number of bugs discovered, and it is convincing if a bug is registered in the database of Common Vulnerabilities and Exposures (**CVE**), and many fuzzing works list discovered CVEs.

Exploitable bugs are vulnerabilities, yet non-exploitable bugs still reflect a fuzzer's performance. Furthermore, reporting vulnerabilities to CVE requires expensive coordination between the vendor and the MITRE Cooperation. One adjustment is listing the **bugs** discovered, but it is still challenging to deduplicate crashes into bugs: many inputs may exercise the same path, and many paths may be affected by the same bug. The ground truth can only be obtained manually: fix the bug, and rerun the test; then inputs that can not crash the program any more are duplicates of the same bug.

However, human intervention for bug confirmation is costly and sometimes infeasible. Therefore, most fuzzing works replace the metric of "bugs discovered" with "**unique crashes**". Instead of manually deduplicate crashes, these works automate the deduplication with heuristics. For instance, AFL's online algorithm `afl-fuzz` deduplicates inputs with significantly similar branch coverage, while its offline algorithm, `afl-cmin`, only keeps inputs covering new branch. In most cases, one unique bug would result in many even hundreds of unique crashes.

The number of bugs are relatively small for most real-world programs, and the number of bugs between different fuzzers is low in contrast. For a given program, there are more **blocks**, **branches** and **paths** than bugs, thus comparing different fuzzers' block-, branch- or path-coverage is also approachable. Covering a block or branch is the prerequisite of detecting the bug residing on it, hence it is also reasonable to use coverage as a metric to evaluate fuzzers.

IV. Industrial Setting Oriented Experiment Setup

In this section, we describe the experiment setup for industrial settings, that running in parallel on large practical projects. During the collaboration with engineers from Alibaba, Huawei and Google, they will not run each fuzzer with single core, which is widely adopted in academic evaluation. They usually dedicate many CPU cores and run the fuzzer in parallel, which allows many fuzzer instances to test the same target at the same time.

A. Prepare Test Suites

The most authentic approach is real bugs from practical large projects from the industry. Originated from Google, **fuzzer-test-suite** [18] is composed of real world libraries, the lines of code for each project ranges from 2948 to 405122, and the average size is 104137 for the 24 practical projects. The suite is carefully crafted for interesting bugs, hard to find paths, or other challenges for bug finding tools. On the contrary, the programs in Google's fuzzer-test-suite is carefully selected and the commit of the source code is precisely defined. Moreover, fuzzer-test-suite ships a test harness for each program, which is guaranteed to trigger a known CVE with proper inputs. The property of known bugs provides the ground truth for evaluation.

To demonstrate the difference between industry-oriented test suites and test suites from the academia, we also include **LAVA-M** [10] in our experiment. It contains four real-world programs from GNU coreutils, and 2265 bugs are injected into the programs. The lines of code for each program ranges from 2064 to 6913. We hope the vast number of known bugs could amplify the minuscule difference in bugs between fuzzers.

B. Execute Tests

Our experiment follows real-world fuzzing practices: run each fuzzer in parallel with existing initial seeds, which is adopted by our industry collaborators from Alibaba, Huawei and Google.

To accelerate fuzzing, the industry dedicates thousands of CPU cores to test programs continuously. Therefore, we run each fuzzer in parallel to reveal their performance under industrial settings. More than just following the industry practice, we also add single-instance experiments for comparison, as the academia commonly evaluate fuzzers in single instance. For fair comparison, both the single run and the parallel run use the corresponding initial seeds.

Due to the characteristics of fuzzers, they are invoked slightly differently but fairly, as listed below:

- Fuzzers of AFL family support syncing, and we run 4 instances of them in parallel: one in master mode, others in secondary mode.
- libFuzzer stops fuzzing on crash. A script ensures 4 instances of libFuzzer by relaunching on crash.
- Radamsa can only generate inputs. A script launches AFL for fuzzing, and invokes Radamsa every 120 seconds for generating new seeds.
- QSYM is tailored for hybrid fuzzing. A script combines a QSYM instance and 3 AFL instances. Note that QSYM is designed for running together with a fuzzer, and we exclude it from the single-instance experiment.

The server is equipped with 32 logic cores (Xeon E5-2630 v3 @ 2.40GHz) and 128 GiB of RAM. The host OS is Ubuntu 16.04 with Linux 4.4.0. Each experiment is given 24 hours to run. Each program in the test suite is instrumented with

Address Sanitizer [28] to immediately signal the fuzzer on latent bugs.

Our experiment covers fuzzers of all types: blackbox, greybox, whitebox and hybrid fuzzers are included. Some of them originate from the industry, while some are written by researchers. Basing on Table I, we select representative fuzzers of each type:

- For blackbox fuzzers, we select **Radamsa** for its unique heuristics to detect the structure of the seed, and mutate with the knowledge accordingly.
- For greybox fuzzers, we select **AFL** and **libFuzzer**, the fuzzers widely used in the industry. We also select **AFLFast** and **FairFuzz**, the representative work from the academia.
- For whitebox fuzzers, we select **KLEE**, which is popular in both the industry and the academia.
- For hybrid fuzzers, we select **QSYM**, which outperforms many state-of-the-art hybrid fuzzing works and even some whitebox fuzzing techniques such as Angora.

C. Collecting Data

Our experiment selects branch count, path count and bugs count as the metrics. After collecting the inputs generated by each fuzzer, we calculate each metric with the methods below:

Bug count is the ultimate evaluation metric for a fuzzer. However, as Section III-B describes, it requires expensive human intervention to gather. We substitute manual verification with stack signature: two crash-leading inputs are considered equivalent if the source locations of all frames match each other. The method above is prone to overestimation, since a bug may be triggered by many paths. To address such problem, we deduplicate the inputs with an Address Sanitizer instrumented binary — program immediately crashes, exactly on the site of memory-unsafe operation. This method significantly reduces the number of duplicates compared to "unique crashes" in AFL and the result of `afl-cmin`.

Branch count directly reflects a fuzzer's ability to cover different states of a binary. However, many fuzzers (e.g. the blackbox fuzzer Radamsa) do not implement this kind of metric in their statistics. For fair comparison, after collecting the inputs generated by each fuzzer, a modified version AFL is invoked on the inputs to count the covered branches.

Path count is a metric similar to branch count. It is more precise, because the calculation is based on the logarithm of the hit count for each branch. The paths of two inputs are considered equivalent if the logarithms of all branches match. To collect this metric, we run AFL on the seeds generated by each fuzzer. In this section, we present the firsthand results and the analysis of the pitfalls. Table III and IV present the path count and bug count of each fuzzer on each project. The first 24 rows are programs in Google's fuzzer-test-suite, and the last 4 rows are programs in LAVA-M. The leftmost 6 columns are fuzzers in single instance, and the rightmost 6 columns are fuzzers in 4 instances in parallel.

V. Results and Lessons Learned

A. Unstable Performance of Academic Fuzzers

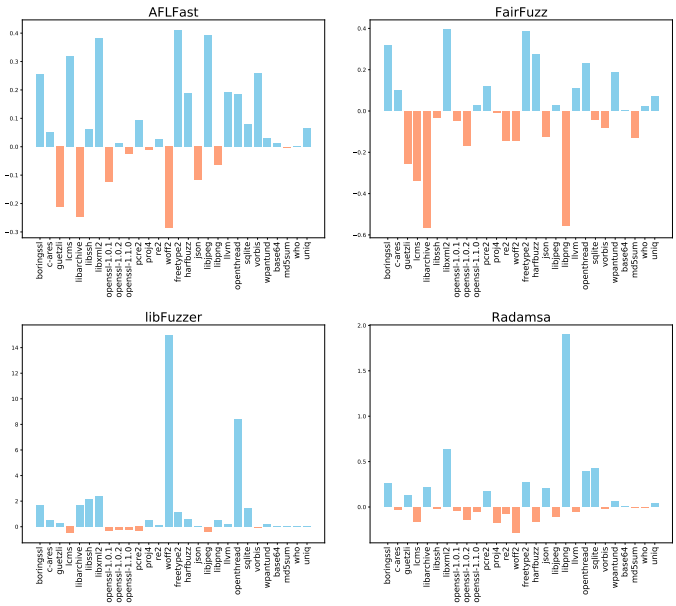

Fig. 1. Variance in Path Count Gains

Table III and Table IV is experiment results also published in [7]. Figure 1 demonstrates the gains of path count over different programs for each fuzzer. The data is generated from Table III as follows: for each program and each fuzzer, we calculate then normalize the gain of path count over AFL.

Inside Figure 1, we can clearly see that the gains of each fuzzer vary over different programs. For each program, if a fuzzer performs as good as AFL, then the gain would be 0%. Better performance is drawn in upward blue bar, while worse performance is drawn in downward red bar. For each fuzzer including descendants of AFL, red bars and blue bars appear alternatively, which indicates that each fuzzer fails to *consistently* perform better than AFL.

Take AFLFast as an example. As Section II-B describes, AFLFast is a variant of AFL which improves the selection stage with power schedule. The effectiveness of power schedule is proved by individual cases of AFLFast — it achieves up to 38% gain in path count. However, when analyzing the gains collectively, the advantage vanishes. Out of all the 28 programs, AFLFast performs worse than AFL on 32% of the programs when considering path count. To sum up, the solid individual gains indicate strong performance, but a significant number of worsened cases indicates unstable performance.

The problem of unstable performance can also be observed in FairFuzz, yet another descent of AFL. Table V summarizes the number of programs each fuzzer performs better or worse than the baseline fuzzer AFL. From the table we can observe that FairFuzz fails to outperform AFL on 50% of the programs when considering path count. This also applies to QSYM and Radamsa, two fuzzers which run collaboratively with AFL.

TABLE III
PATH COUNT

	AFL	AFLFast	FairFuzz	libFuzzer	Radamsa	KLEE	4 QSYM	4 AFL	4 AFLFast	4 FairFuzz	4 libFuzzer	4 Radamsa
boringssl	1334	1674	1760	**3528**	1682	42	1207	3286	2816	3393	**5525**	3430
c-ares	80	84	88	**123**	78	25	72	146	116	146	**191**	146
guetzli	1382	1090	1030	**1773**	1562	103	1268	3248	2550	1818	**3844**	3342
lcms	656	**864**	434	338	550	2	605	**1682**	1393	1491	1121	1416
libarchive	3756	2834	1630	**10124**	4570	82	3505	12842	10111	12594	**22597**	12953
libssh	64	68	62	**201**	63	4	87	110	102	110	**362**	110
libxml2	5762	7956	8028	**19663**	9392	18	5098	14888	13804	14498	**28797**	17360
openssl-1.0.1	**2397**	2103	2285	1709	2303	N/A	2330	**3992**	3501	3914	2298	3719
openssl-1.0.2	2456	**2482**	2040	1881	2108	N/A	1947	**4090**	3425	3956	2304	3328
openssl-1.1.0	2439	2380	**2501**	1897	2311	N/A	2416	4051	3992	**4052**	2638	3593
pcre2	32310	35288	36176	20981	**37850**	261	24501	**79581**	66894	71671	59616	78347
proj4	220	218	218	**334**	182	24	208	342	302	322	**509**	341
re2	5860	6014	5016	**6327**	5418	267	5084	12093	10863	12085	**15682**	12182
woff2	14	10	12	**224**	10	N/A	15	23	16	20	**447**	22
freetype2	7748	10939	10714	**16360**	9825	N/A	7188	19086	18401	20655	**25621**	18609
harfbuzz	6793	8068	8668	**10800**	5688	N/A	6881	12398	11141	14381	**16771**	11021
json	466	412	408	499	**564**	326	504	1096	963	721	1081	**1206**
libjpeg	704	**979**	722	448	634	85	638	1805	1579	**2482**	1486	1632
libpng	170	159	76	263	**493**	2	577	582	568	**587**	586	547
llvm	4830	**5760**	5360	5646	4593	N/A	4096	8302	8640	9509	**10169**	8019
openthread	104	123	127	**976**	144	N/A	141	268	213	230	**1429**	266
sqlite	179	193	172	**431**	256	88	180	298	322	294	**580**	413
vorbis	891	**1122**	821	848	875	11	898	1484	1548	**1593**	1039	1381
wpantund	2959	3048	**3513**	3510	3146	N/A	2975	4914	5112	**5691**	4881	4891
base64	1060	**1072**	1061	N/A	1061	342	**1643**	1078	1065	1080	N/A	1077
md5sum	**584**	582	508	N/A	583	134	**1062**	589	589	601	N/A	589
who	4397	4403	**4497**	N/A	4367	14	**5621**	4599	4585	4593	N/A	4415
uniq	420	447	**450**	N/A	437	427	**693**	476	453	471	N/A	480

Different from AFL's branch metric, libFuzzer guides fuzzing by basic block, which could explain the derivation in coverage-related metrics. This does not imply that libFuzzer is inferior to AFL — out of 46% programs, it discovers more bugs.

Lesson 1: Traditionally, researchers aim at improving the *performance* of fuzzing techniques, i.e. better result on a specific set of programs. Our results reveal that, it is equally important to improve the *stability*, i.e. consistent performance improvements over general programs.

B. Impaired Performance on Parallel Runs

Figure 2 illustrate the path count of 4 instances of fuzzer running in parallel. Similar to Section V-A, the data is generated by normalizing the metric recorded in 4 instances with the metric of 4 AFLs. Similar to Figure 1, if a fuzzer performs as good as 4 AFLs for a program, then the gain would be 0%. Better performance is drawn in upward blue bar, while worse performance is drawn in downward red bar.

Compare the gains in path count of a single instance (as in 1) and 4 instances of fuzzers running in parallel (as in Figure 2), we can see that different fuzzers have drastically different performance characteristics when running in single or parallel. Most fuzzers which perform well in one instance fail to maintain the improvement in four instance runs.

For instance, AFLFast in one instance has an average gain of 7% in path count, but the average gain of four instance is -9%. Analyzing performance of individual cases yields the same conclusion: it outperforms AFL in 68% of programs when considering path coverage, and in a total of 4 programs

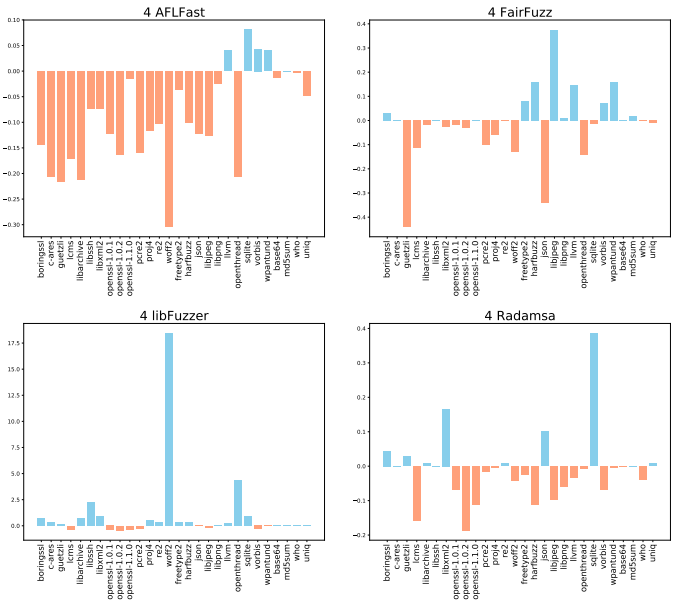

Fig. 2. Path Count Gains of 4 Instances

does it discover more bugs. However, when running in four instances, the situation is quite the opposite — the number of programs it outperforms AFL collapses to 14% in path coverage. Coverage drops, and an obvious result is diminished bug count: only in one program does it outperform AFL, and it discovers less bugs in 36% of programs.

The problem of impaired performance on parallel runs is not limited to AFLFast. For instance, the number of programs

TABLE IV
BUG COUNT

	AFL	AFLFast	FairFuzz	libFuzzer	Radamsa	KLEE	4 QSYM	4 AFL	4 AFLFast	4 FairFuzz	4 libFuzzer	4 Radamsa
boringssl	0	0	0	**1**	0	0	0	0	0	0	**1**	0
c-ares	1	**2**	**2**	1	**2**	1	1	**3**	2	**3**	1	2
guetzli	0	0	0	0	0	0	0	0	0	0	**1**	0
lcms	0	0	0	0	0	0	0	1	1	1	**2**	1
libarchive	0	0	0	0	0	0	0	0	0	0	**1**	0
libssh	0	0	0	**1**	0	0	0	0	0	0	**1**	0
libxml2	0	**1**	0	**1**	**1**	0	0	1	1	1	**3**	2
openssl-1.0.1	0	0	0	0	0	N/A	0	**3**	2	**3**	2	2
openssl-1.0.2	**2**	1	0	1	1	N/A	2	**5**	4	4	1	**5**
openssl-1.1.0	0	0	0	0	0	N/A	0	**5**	**5**	**5**	3	4
pcre2	**2**	1	1	1	**2**	0	1	**6**	4	5	2	5
proj4	0	0	0	**1**	0	0	0	**2**	0	1	1	1
re2	0	0	0	**1**	0	0	0	**1**	0	**1**	**1**	0
woff2	0	0	0	**1**	0	N/A	0	1	0	0	**2**	1
freetype2	0	0	0	0	0	N/A	0	0	0	0	0	0
harfbuzz	0	0	0	**1**	0	N/A	0	0	0	**1**	**1**	0
json	**1**	**1**	0	0	**1**	0	0	2	1	0	1	**3**
libjpeg	0	0	0	0	0	0	0	0	0	0	0	0
libpng	0	**1**	**1**	**1**	**1**	0	**1**	0	0	0	0	0
llvm	0	0	**1**	**1**	0	N/A	1	1	1	**2**	**2**	1
openthread	0	0	0	**1**	0	N/A	0	0	0	0	**4**	0
sqlite	0	0	0	**1**	**1**	0	1	0	0	0	**3**	1
vorbis	1	1	**2**	1	1	0	2	3	**4**	3	3	3
wpantund	0	0	0	0	0	N/A	0	0	0	0	0	0
base64	0	**1**	0	N/A	0	0	**41**	1	1	0	N/A	0
md5sum	0	0	0	N/A	0	0	**57**	0	0	1	N/A	0
who	0	0	0	N/A	0	0	**1047**	2	0	1	N/A	0
uniq	5	5	6	N/A	5	**8**	**25**	11	5	7	N/A	5

TABLE V
PERFORMANCE GAINS OR LOSES, PROGRAM-WISE COUNTS

Fuzzer	Path+	Path-	Branch+	Branch-	Bug+	Bug-
AFLFast	**19**	9	**16**	10	4	2
FairFuzz	14	14	14	10	5	3
libFuzzer	17	7	11	13	**11**	3
Radamsa	13	15	15	7	4	1
KLEE	1	**18**	2	**17**	1	**18**
4 QSYM	13	15	**12**	15	7	3
4 AFLFast	4	**23**	4	**21**	1	**10**
4 FairFuzz	11	15	9	15	3	8
4 libFuzzer	**15**	9	9	15	**11**	7
4 Radamsa	8	17	8	15	3	9

in which FairFuzz fails to keep the advantage over AFL in parallel mode is 3, 5 and 2 when considering path count, branch count and bug count. The same applies to Radamsa, and the number of programs failed to keep the advantage is 5, 7 and 1 respectively. All fuzzers under test lose their advantage to some extent when scaling to 4 instances. Even the industry-originated libFuzzer still fails to keep up the advantage in 2, 2 and 1 programs respectively. The best performing fuzzer under test is AFL, a fuzzer heavily optimized with feedback from the industry. It is common to run fuzzers in parallel in industrial environments. For example, Google dedicates over 25,000 cores just to fuzz ˜200 projects [1]. The industry heavily optimizes the scalability of a fuzzer, and related techniques such as core-binding and in-process fuzzing mostly originates from the industry. Based on the results, we can learn the following lesson.

Lesson 2: Traditionally, researchers are used to evaluate their works in *single* instance. Our results reveal that some strategies achieve good performances in single instance settings while underperform when running in parallel. To evaluate the industrial applicability, it is also important to extent experiments on running fuzzing techniques in *parallel*.

C. Bottleneck of Symbolic Execution

Figure 3 demonstrates the bottleneck of symbolic execution on large scale programs. The data is generated from III as follows: we first calculate the gain QSYM over AFL for each program, and then normalize the gain by dividing it by the corresponding metric of 4 AFL. For KLEE, the metric is normalized with single AFL. The programs are sorted by lines of code in ascending order.

Inside the first row of Figure 3, we can frequently see long blue bars in the left side. That is to say, QSYM significantly improves the performance on programs with less lines of code (2k – 6k). Except for the rightmost outlier, many bars in the middle are red ones or short blue ones, which indicates that QSYM fails to maintain its advantage on other programs with a large codebase. The same applies to KLEE, as the branch counts is generally lower on larger programs. If we focus on programs in LAVA-M, an interesting pattern emerges. All the 4 programs in LAVA-M — `uniq`, `base64`, `md5sum` and `who` — are among the top 6 smallest programs. Among all the programs in LAVA-M, QSYM dominates all the metric: average gains of 50%, 337% in path count and branch count are achieved; a total of 1170 bugs are discovered, which is a vast

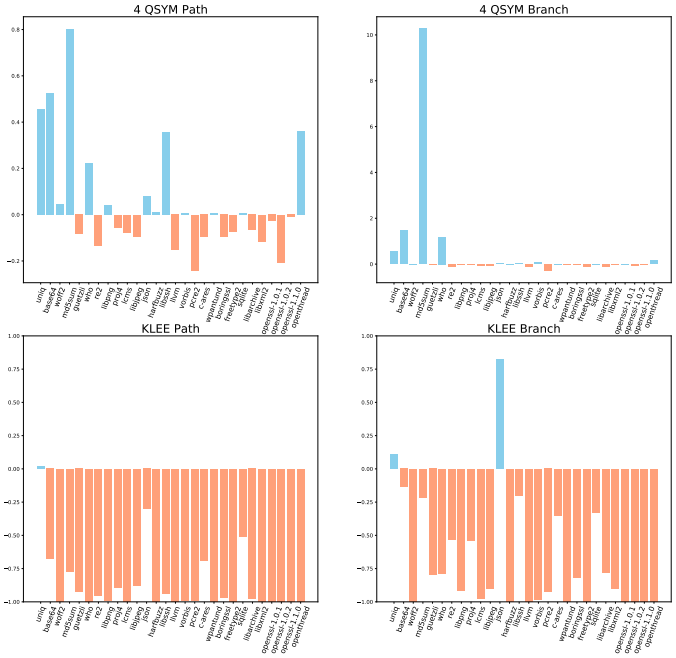

Fig. 3. Gains of QSYM and KLEE (Sorted by LoC in Ascending Order)

8257% improvement over AFL. However, if such programs are excluded, QSYM fails to maintain its dominance on real-world projects: only on `openthread` does it significantly outperform AFL in all metrics, and the average gains of path count and branch count both drop to -3%. A -3% gain is also observed in total number of bugs discovered. Analyzing the results program-wise, we can draw the same conclusion: out of the 24 programs excluding LAVA-M, QSYM fails to outperform AFL on 63% of programs when considering path coverage or branch coverage, and 13% of the programs when considering bug count.

As a state-of-the-art implementation of symbolic execution, QSYM aggressively optimizes its performance even at the cost of considerable implementation complexity. The outlier does demonstrate the strengths of symbolic execution. For example, `openthread` is an implementation of various network protocols, and checksum is ubiquitous in protocols. The checksum is almost impossible to be guessed correctly, but it is trivial for symbolic execution engines.

However, it still fails to outperform AFL in real-world programs with large codebase. The reason behind may relate to the size of the program. It is well known that symbolic execution techniques suffer from path explosion. In other words, the number of possible paths to explore grows exponentially to the number of branches. Hybrid fuzzing works alleviate the problem by offloading the "trivial" part of exploration to fuzzers, and only solve the "difficult" part. Therefore, valueless paths can be discarded by the symbolic execution engine, and the problem of path explosion is alleviated. However, the statistics from fuzzing are not a silver bullet, especially when the program gets larger. Exponential path growth is the intrinsic of symbolic execution, and no optimization lowers the asymptotic computation complexity, not to mention the stage of SMT solving, an NP-complete problem. Given a program that is large enough, the performance gains brought by symbolic execution tend to diminish.

> **Lesson 3:** Recently, a popular approach of hybrid fuzzing is combining symbolic execution with fuzzing, and its effectiveness on *small programs* such as LAVA-M suite and CGC suite is impressive. Our results reveal that, symbolic execution may have bottleneck on *large codebase*. For industrial usage, it is recommanded to evaluate hybrid fuzzing on real-world programs such as almost all practical projects from Google's fuzzer-test-suite.

VI. Implications and Solutions

In this section, we highlight implications and potential solutions to benefit fuzzing in industrial settings. As Section V analyses, existing fuzzing works may face some pitfalls when running in industrial environment, such as unstable performance, impaired parallel performance, and bottleneck of symbolic execution. The general idea of our potential solutions is to vertically or horizontally orchestrate multiple dynamic testing techniques (e.g. various fuzzing strategies, symbolic execution engines, taint analyzers) to avoid pitfalls of single technique. Fig 4 and 5 show the high level designs of horizontal orchestration and vertical orchestration. Horizontal orchestration combines multiple dynamic testing techniques in parallel by synchronizing statistics inside each technique. Custom implementation is allowed by considering what techniques should be chosen, what statistics should be synchronized and how statistics are synchronized. Vertical orchestration switches among multiple dynamic testing techniques in series. The key point of it is switch strategy, which determines what to switch and when to switch. The solutions we propose to strengthen existing fuzzing works in industrial settings is based on above two orchestrations.

A. Improving Performance Stability

As Section V-A analyzes, researchers target improving the performance of fuzzing techniques, but it is equally important to improve the stability, i.e. consistent performance improvements over general programs. Note that even for the fuzzer with the best overall performance, other fuzzers still excel its performance on some programs. One explanation is that each fuzzer has unique strengths and performs the best on some particular programs. It is conceivable that the orchestration of different fuzzers results is more stable. To evaluate the performance of the preliminary study, we select 10 programs in pure C from Google's fuzzer-test-suite, on which existing fuzzers perform most variously.

To improve stability, we propose HoFuzz, an implementation of horizontal orchestration. HoFuzz runs different fuzzers in parallel while sharing seeds between each other real time. Figure 4 depicts the performance of HoFuzz. In this scheme,

HoFuzz, allocates 4 cores: 2 cores for AFL, one core for FairFuzz and one core for AFLFast. Each fuzzer shares its seeds by the built-in periodical synchronization of AFL. Equally given 4 cores, the performance of pure FairFuzz or pure AFLFast varies; but only HoFuzz consistently performs better than pure AFL, AFLFast and FairFuzz. **The proposed method could cover 36%-82% more paths, trigger 13%-41% more branches and discover 46%-150% more unique crashes or bugs, compared to fuzzers such as AFL, FairFuzz and QSYM.**

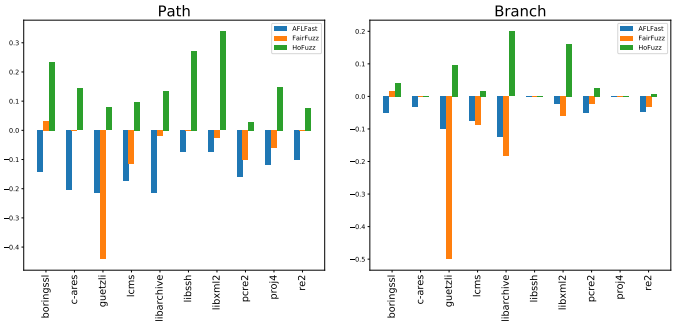

Fig. 4. Gains of Horizontal Orchestration

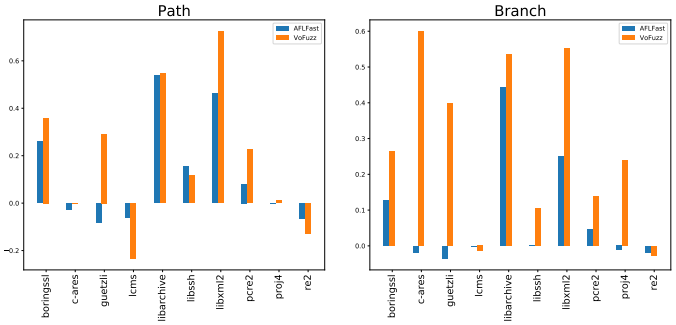

Fig. 5. Gains of Vertical Orchestration

We also propose VoFuzz, an implementation of vertical orchestration. In this scheme, the heuristics of FairFuzz is enhanced with AFLFast. The rationale is that FairFuzz's strategy only optimizes for situations where rarely hit branches exist, but when such branches disappear it will be "stuck". VoFuzz prioritizes the strategy of FairFuzz, but switches to AFLFast's more general strategy when needed. Figure 5 depicts the performance of VoFuzz, an implementation of vertical orchestration. VoFuzz also shows better stability than the existing fuzzing techniques alone.

Further research may focus on fine-tuning the orchestration to improve the stability of performance. For example, in the scenario of horizontal orchestration, the optimal combination of fuzzers on different number cores may be inferred beforehand; in the scenario of vertical orchestration, the strategy to switch between different strategies remains to be explored.

B. Optimizing Parallel Performance

As Section V-B analyzes, some optimization strategies from the academia achieve good performances in single instance settings while underperform when running in parallel. One of the reasons could be the statistical data behind the evolution of "dumb fuzzers". Usually, "dumb" fuzzers such as AFL mutate the seeds blindly. With little connection between each fuzzer, the performance characteristics of "dumb" fuzzers running in parallel is similar to which in standalone mode. However, as fuzzers become smarter and smarter, statistical data have became a critical part of a "smart" fuzzer. If not specifically designed for parallel runs, only a local view of the data is available to each fuzzer. Without an unbiased view statistical data, the scheduler may guide fuzzing to local optimum instead of global optimum. For example, FairFuzz proposes heuristics that only keeps seeds exercising low-frequency branches. If a fuzzer can not share the count of branch hits with each other, a branch hit by other fuzzers many times may be considered low-frequency branch in current fuzzer.

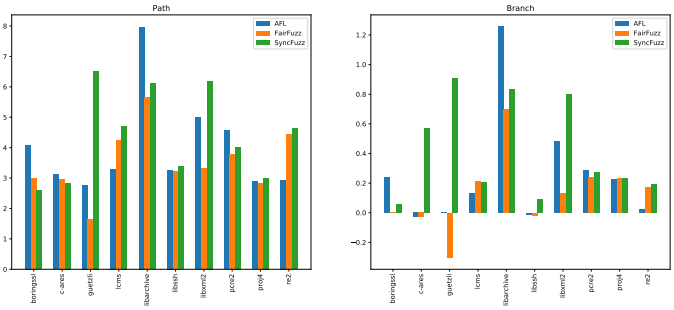

Fig. 6. Gains of Synchronization

For industrial fuzzers running in parallel, it is a de facto practice to "steal" seeds from other instance of fuzzers. This idea also applies when scaling fuzzers which leverages statistical information. The implementation is slightly different, as fuzzers not only passively accept the statistics, but also actively update them. To scale FairFuzz, we propose SyncFuzz, an implementation of Horizontal orchestration. It creates a global storage of such information and update it in each fuzzer collaboratively. Figure 6 depicts the performance gains when statistics are shared between different instance of fuzzers. **Augmented with the proposed information synchronization during parallel runs, AFLFast and FairFuzz could cover 8% and 17% more branches, and trigger 79% and 52% more unique crashes.**

Further research may focus on optimizing parallel performance with effective synchronization. For example, the contention of locks in process creation could be mitigated, and the seeds could be more effectively synchronized and distributed for large-scale fuzzing.

C. Balancing Symbolic Execution

As Section V-C analyzes, state-of-the-art symbolic execution techniques usually decelerate fuzzing on large-scale real-

world programs. Sophisticated as QSYM is, only on programs sensitive to magic numbers does it perform significantly better than AFL in both path count and branch count.

Hybrid fuzzing is a special case of vertical orchestration, so the switch strategy significantly impacts performance. The key of hybrid fuzzing is balancing fuzzing and symbolic execution. Occasionally switching to symbolic execution greatly avoids the overhead of symbolic execution, but our results indicate that the strategy still stresses symbolic execution. It is well known that symbolic execution is a heavyweight approach. The cost of tracing greybox fuzzing, collecting constraints and solving constraints still exist, even though the number of invocation is reduced by hybrid fuzzing.

We speculate that hybrid fuzzing could balance two techniques better, by adopting a more lightweight approach when invoking symbolic execution. To further save the cost of symbolic execution, we propose moving the symbolic execution forward. Specifically, we run KLEE as the symbolic execution engine for one hour, and then collect the results as the initial seeds for fuzzing; next, we run AFL for 23 hours with the aforementioned seeds. In this scenario, symbolic execution only generates meaningful seeds, instead of running in parallel with fuzzing. In this way, we can control the time and resource usage and eliminate the cost of switching between two techniques.

For example, `pcre2` is the program where QSYM has the worst performance gain compared to AFL. **QSYM achieves a -24% gain in path count, a -28% gain in branch count, and a -32% gain in bug count**. Figure 7 compares the heavyweight approach of QSYM and our lightweight approach on `pcre2` using path count and branch count. **The proposed lightweight approach constantly stays atop in each metric, and finally outperforms by 3% in both branch count and path count with less computation resource consumption.** Sometimes, running pure fuzzing on practical large projects would be better than augmented with symbolic execution, as demonstrated in Table V.

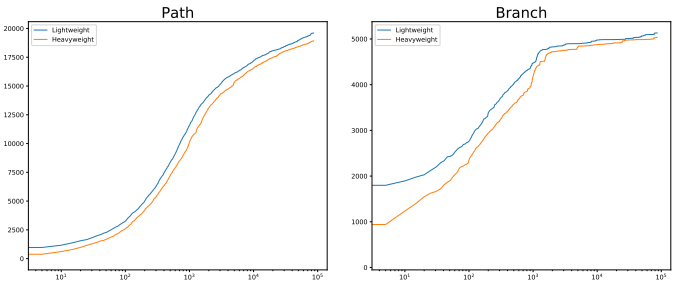

Fig. 7. Performance of Lightweight and Heavyweight Symbolic Execution

Further research could balance symbolic execution and fuzzing with better heuristics, such as ignoring seeds stuck at unimportant branches with static analysis. However, this approach only partially alleviates the deficiency of symbolic execution. Looking forward, breakthroughs in symbolic execution and SMT solver could greatly benefit hybrid fuzzing.

VII. Conclusion

This paper test the existing fuzzing techniques to reveal their performances in industrial settings, that run fuzzers in parallel on large piratical projects. We first give a brief introduction to existing fuzzing works, and then review test suites and metrics commonly used by the academia. Basing on the analysis of strengths and weaknesses, we systematically select Google's fuzzer-test-suite from industry and LAVA-M suite from academia. The experiment environment is strictly controlled to mimic industrial environment.

The experiment results reveal some pitfalls of existing fuzzing works in industry settings, such as unstable performance, impaired parallel performance, and bottleneck of symbolic execution. We propose two general frameworks, namely horizontal orchestration and vertical orchestration, to tackle these pitfalls. For each pitfall, we analyze the root causes, envision potential solutions based on the general frameworks and prove their effectiveness with preliminary experiments. To improve stability of performance, different fuzzers may be orchestrated vertically or horizontally. To optimize parallel performance, statistics may be synchronized among instances. To overcome the bottleneck of symbolic execution, symbolic execution may be invoked in a more lightweight approach. Those industrial oriented optimizations can be leveraged for more scalable performance.

Acknowledgement

This research is sponsored in part by National Key Research and Development Project (Grant No. 2019YFB1706200, 2018YFB1703404), the NSFC Program (No. 62022046, U1911401, 61802223), the Huawei-Tsinghua Trustworthy Research Project (No. 20192000794), and the Beijing Municipal Science & Technology Commission (Project Number: Z191100007119010).

References

[1] Abhishek Arya et al. *Open sourcing ClusterFuzz*. 2019. URL: https://opensource.googleblog.com/2019/02/open-sourcing-clusterfuzz.html (visited on 04/28/2019).

[2] Thanassis Avgerinos et al. "AEG: Automatic Exploit Generation". In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011. URL: http://www.isoc.org/isoc/conferences/ndss/11/pdf/5_5.pdf.

[3] 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, Vienna, Austria, October 24-28, 2016*. Ed. by Edgar R. Weippl et al. ACM, 2016, pp. 1032–1043. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978428. URL: https://doi.org/10.1145/2976749.2978428.

[4] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. Ed. by Richard Draves and Robbert van Renesse. USENIX Association, 2008, pp. 209–224. URL: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

[5] Sang Kil Cha, Maverick Woo, and David Brumley. "Program-Adaptive Mutational Fuzzing". In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 725–741. ISBN: 978-1-4673-6949-7. DOI: 10.1109/SP.2015.50. URL: https://doi.org/10.1109/SP.2015.50.

[6] Peng Chen and Hao Chen. "Angora: Efficient Fuzzing by Principled Search". In: *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 2018, pp. 711–725. DOI: 10.1109/SP.2018.00046. URL: https://doi.org/10.1109/SP.2018.00046.

[7] Yuanliang Chen et al. "EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers". In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. 2019, pp. 1967–1983. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/chen-yuanliang.

[8] Yuanliang Chen et al. "EnFuzz: From Ensemble Learning to Ensemble Fuzzing". In: *CoRR* abs/1807.00182 (2018). arXiv: 1807.00182. URL: http://arxiv.org/abs/1807.00182.

[9] *Document management – Portable document format – Part 2: PDF 2.0*. Standard. Geneva, CH: International Organization for Standardization, July 2017.

[10] Brendan Dolan-Gavitt et al. "LAVA: Large-Scale Automated Vulnerability Addition". In: *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 2016, pp. 110–121. DOI: 10.1109/SP.2016.15. URL: https://doi.org/10.1109/SP.2016.15.

[11] Will Dormann et al. *CERT BFF - Basic Fuzzing Framework*. URL: https://vuls.cert.org/confluence/display/tools/CERT+BFF+-+Basic+Fuzzing+Framework (visited on 03/07/2019).

[12] Dustin Fraze. *Cyber Grand Challenge (CGC)*. 2015. URL: https://www.darpa.mil/program/cyber-grand-challenge (visited on 03/18/2019).

[13] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. "Automated Whitebox Fuzz Testing". In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. The Internet Society, 2008. URL: http://www.isoc.org/isoc/conferences/ndss/08/papers/10_automated_whitebox_fuzz.pdf.

[14] Google. *Fuzzer Test Suite*. URL: https://opensource.google.com/projects/fuzzer-test-suite (visited on 03/07/2019).

[15] Aki Helin. *Aki Helin / radamsa*. URL: https://gitlab.com/akihe/radamsa (visited on 03/07/2019).

[16] Sam Hocevar. *zzuf - multi-purpose fuzzer*. URL: http://caca.zoy.org/wiki/zzuf (visited on 03/07/2019).

[17] Allen Householder. *CERT Triage Tools*. URL: https://vuls.cert.org/confluence/display/tools/CERT+Triage+Tools (visited on 03/07/2019).

[18] Google Inc. *google/fuzzer-test-suite: Set of tests for fuzzing engines*. 2016. URL: https://github.com/google/fuzzer-test-suite/ (visited on 03/18/2019).

[19] George Klees et al. "Evaluating Fuzz Testing". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by David Lie et al. ACM, 2018, pp. 2123–2138. DOI: 10.1145/3243734.3243804. URL: https://doi.org/10.1145/3243734.3243804.

[20] Caroline Lemieux and Koushik Sen. "FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. Ed. by Marianne Huchard, Christian Kästner, and Gordon Fraser. ACM, 2018, pp. 475–485. DOI: 10.1145/3238147.3238176. URL: https://doi.org/10.1145/3238147.3238176.

[21] Jie Liang et al. "DeepFuzzer: Accelerated Deep Greybox Fuzzing". In: *IEEE Transactions on Dependable and Secure Computing* (2019).

[22] Jie Liang et al. "Engineering a better fuzzer with synergically integrated optimizations". In: *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2019, pp. 82–92.

[23] Jie Liang et al. "PAFL: extend fuzzing optimizations of single mode to industrial parallel mode". In: *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. Ed. by Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu. ACM, 2018, pp. 809–814. ISBN: 978-1-4503-5573-5. DOI: 10.1145/3236024.3275525. URL: https://doi.org/10.1145/3236024.3275525.

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

[25] LLVM Project. *libFuzzer – a library for coverage-guided fuzz testing.* URL: https://llvm.org/docs/LibFuzzer.html (visited on 03/07/2019).

[26] Sanjay Rawat et al. “VUzzer: Application-aware Evolutionary Fuzzing”. In: *NDSS*. The Internet Society, 2017.

[27] Sergej Schumilo et al. “kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels”. In: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. Ed. by Engin Kirda and Thomas Ristenpart. USENIX Association, 2017, pp. 167–182. URL: https://www.usenix.org/conference/usenixsecurity17 / technical - sessions / presentation / schumilo.

[28] Konstantin Serebryany et al. “AddressSanitizer: A Fast Address Sanity Checker”. In: *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*. Ed. by Gernot Heiser and Wilson C. Hsieh. USENIX Association, 2012, pp. 309–318. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[29] Bhargava Shastry et al. “Static Program Analysis as a Fuzzing Aid”. In: *RAID*. Vol. 10453. Lecture Notes in Computer Science. Springer, 2017, pp. 26–47.

[30] Nick Stephens et al. “Driller: Augmenting Fuzzing Through Selective Symbolic Execution”. In: *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016. URL: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf.

[31] Mingzhe Wang et al. “SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing”. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. Ed. by Michel Chaudron et al. ACM, 2018, pp. 61–64. DOI: 10.1145/3183440.3183494. URL: https://doi.org/10.1145/3183440.3183494.

[32] Insu Yun et al. “QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing”. In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, 2018, pp. 745–761. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/yun.

[33] Michal Zalewski. *American fuzzy lop*. URL: http://lcamtuf.coredump.cx/afl/ (visited on 01/18/2019).

[34] Chijin Zhou et al. “Zeror: Speed Up Fuzzing with Coverage-sensitive Tracing and Scheduling”. In: *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2020, pp. 858–870.