Evaluating and Improving Hybrid Fuzzing

Ling Jiang†
Southern University of Science and Technology
Shenzhen, China
11711906@mail.sustech.edu.cn

Hengchen Yuan
Southern University of Science and Technology
Shenzhen, China
11911202@mail.sustech.edu.cn

Mingyuan Wu
Southern University of Science and Technology
Shenzhen, China
11849319@mail.sustech.edu.cn

Lingming Zhang
University of Illinois Urbana-Champaign
Champaign, USA
lingming@illinois.edu

Yuqun Zhang*
Southern University of Science and Technology
Shenzhen, China
zhangyq@sustech.edu.cn

Abstract*—To date, various hybrid fuzzers have been proposed for maximal program vulnerability exposure by integrating the power of fuzzing strategies and concolic executors. While the existing hybrid fuzzers have shown their superiority over conventional coverage-guided fuzzers, they seldom follow equivalent evaluation setups, e.g., benchmarks and seed corpora. Thus, there is a pressing need for a comprehensive study on the existing hybrid fuzzers to provide implications and guidance for future research in this area. To this end, in this paper, we conduct the first extensive study on state-of-the-art hybrid fuzzers. Surprisingly, our study shows that the performance of existing hybrid fuzzers may not well generalize to other experimental settings. Meanwhile, their performance advantages over conventional coverage-guided fuzzers are overall limited. In addition, instead of simply updating the fuzzing strategies or concolic executors, updating their coordination modes potentially poses crucial performance impact of hybrid fuzzers. Accordingly, we propose CoFuzz to improve the effectiveness of hybrid fuzzers by upgrading their coordination modes. Specifically, based on the baseline hybrid fuzzer QSYM, CoFuzz adopts *edge-oriented scheduling* to schedule edges for applying concolic execution via an online linear regression model with Stochastic Gradient Descent. It also adopts *sampling-augmenting synchronization* to derive seeds for applying fuzzing strategies via the interval path abstraction and John walk as well as incrementally updating the model. Our evaluation results indicate that CoFuzz can significantly increase the edge coverage (e.g., 16.31% higher than the best existing hybrid fuzzer in our study) and expose around 2X more unique crashes than all studied hybrid fuzzers. Moreover, CoFuzz successfully detects 37 previously unknown bugs where 30 are confirmed with 8 new CVEs and 20 are fixed.

I. Introduction

Fuzzing usually refers to automated test input generation for exposing potential software bugs or security vulnerabilities. To date, many existing fuzzers facilitate vulnerability exposure by optimizing code coverage of programs under test (i.e., coverage-guided fuzzing [1]–[6]). However, they have been shown ineffective in many occasions [7]–[9]. To address such issue, hybrid fuzzing [10]–[16] has been proposed to augment fuzzing effectiveness by coordinating fuzzing strategies and concolic execution [17]. Specifically, hybrid fuzzers leverage fuzzing strategies to promptly explore program states and concolic executors to generate the inputs which advance in exploring hard-to-cover branches by solving program path constraints. Moreover, hybrid fuzzers develop coordination modes [14]–[16] to schedule subjects to be solved by concolic execution and synchronize the resulting solutions for executing fuzzing strategies to strengthen their effectiveness.

Although hybrid fuzzers have shown their performance superiority over conventional coverage-guided fuzzers, e.g., Angora outperforms AFL by 27.08% in terms of edge coverage in the original paper [11], they seldom follow equivalent evaluation setups, e.g., they hardly perform evaluations on identical benchmarks or initial seed corpora. For instance, QSYM [10] and Eclipser [12] adopt no common benchmark programs for their evaluations. Meanwhile, the performance comparisons among the existing hybrid fuzzers are also limited. For instance, Intriguer [13] only has been evaluated against QSYM [10] in the existing literature. Such inconsistent evaluation setups and limited performance comparisons can potentially compromise the effectiveness and reliability of the existing hybrid fuzzers. Therefore, there is a pressing need for an extensive study on the existing hybrid fuzzers to comprehensively delineate their strengths, limitations, and rationale.

In this paper, to our best knowledge, we conduct the first comprehensive study on the existing hybrid fuzzers. Specifically, we select seven state-of-the-art hybrid fuzzers as our study subjects and construct a comprehensive benchmark suite with 15 commonly adopted programs in their original papers. Our study results suggest that the performance of existing hybrid fuzzers may not well generalize to other experimental setups. For instance, while Intriguer [13] and MEUZZ [15] outperform QSYM in the original papers, our study shows that QSYM can outperform Intriguer and MEUZZ by 9.69% and

† Ling Jiang is also affiliated with the Research Institute of Trustworthy Autonomous Systems, Shenzhen, China.

* Yuqun Zhang is the corresponding author. He is also affiliated with the Research Institute of Trustworthy Autonomous Systems, Shenzhen, China and Guangdong Provincial Key Laboratory of Brain-inspired Intelligent Computation, China

6.29% in terms of edge coverage respectively on our more comprehensive benchmark suite. Meanwhile, while hybrid fuzzers can overall outperform conventional coverage-guided fuzzers, the performance advantage is somewhat limited or even untenable. For instance, on project *tcpdump*, conventional coverage-guided fuzzer AFL++ outperforms hybrid fuzzers Intriguer and MEUZZ by 7.02% and 4.77% respectively. Moreover, most studied hybrid fuzzers only expose slightly more or even fewer unique crashes than conventional coverage-guided fuzzers, e.g., AFL exposes 109 crashes while Eclipser and DigFuzz only expose 107 and 105 crashes respectively. We further find that the coordination mode can be a key factor for the performance impact of a hybrid fuzzer, while the power of the existing seed-oriented scheduling mechanisms and the synchronization mechanisms have not been fully leveraged.

Inspired by the findings of our study, we propose a novel hybrid fuzzing framework named CoFuzz which improves the coordination mode upon the fuzzing strategies and concolic executor of the baseline hybrid fuzzer QSYM. In particular, CoFuzz applies *edge-oriented scheduling* which adopts an online linear regression model with Stochastic Gradient Descent [18] to schedule hard-to-cover edges to be solved by concolic executor. Furthermore, CoFuzz adopts *sampling-augmenting synchronization* to derive the seeds with the interval abstraction domain [19] and John walk [20] for executing the fuzzing strategy. Our evaluation results indicate that CoFuzz can outperform AFL and the top-performing hybrid fuzzer QSYM by 32.44% and 16.31% respectively in terms of edge coverage. Meanwhile, CoFuzz successfully exposes around 2X more unique crashes compared with state-of-the-art hybrid fuzzers. Moreover, CoFuzz can detect 37 previously unknown bugs where 30 have been confirmed with 8 new CVEs and 20 have been fixed.

To summarize, our paper makes the following contributions.

- We have performed an extensive study on state-of-the-art hybrid fuzzers on 15 real-world open-source projects widely used in prior work. We find that their performance may not well generalize to other experimental settings and their performance advantages over conventional coverage-guided fuzzers are overall limited. Moreover, improving the coordination mode can be a key factor to augment the performance of hybrid fuzzers.
- We propose a hybrid fuzzing framework CoFuzz based on our findings which can significantly outperform the best existing hybrid fuzzer by 16.31% in terms of edge coverage and expose 37 previously unknown bugs which cannot be detected by any studied hybrid fuzzer.

Note that all our study details are presented in our GitHub repository [21].

II. Background

A hybrid fuzzer [10], [22]–[24] typically consists of a coverage-guided fuzzing strategy and a concolic executor, and coordinates them via a coordination mode including the scheduling and synchronization mechanism. Note that although other implementations of coordination modes may exist, in this paper, we follow various prior work [25]–[29] to select the representative mode. Figure 1 presents a typical framework of a hybrid fuzzer where the coverage-guided fuzzing strategy iteratively obtains a seed from the seed corpora and generates the mutants as input to the program under test (PUT). By acquiring the code coverage updates caused by executing mutants, the fuzzing strategy retains the ones with increased/optimized code coverage as the seeds in the seed corpora for further mutations. Meanwhile, the collected seeds from the seed corpora are also scheduled for applying concolic execution to generate the results, i.e., in essence deriving the mutants, by solving path constraints pc via the SMT solver [30]. Next, the resulting mutants after applying concolic execution are synchronized, i.e., input for executing the fuzzing strategy. In this section, we present the details of coverage-guided fuzzing strategy, concolic executor, and coordination mode of hybrid fuzzers respectively.

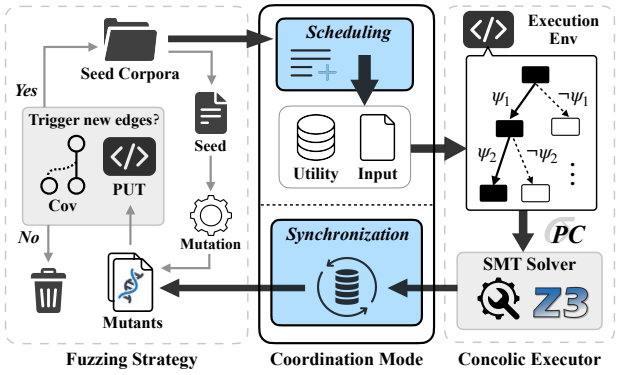

Fig. 1. The overall framework of hybrid fuzzing

A. Coverage-guided Fuzzing Strategy

Coverage-guided fuzzers [1]–[3] usually maximize/increase code coverage for advancing iterative executions (e.g., mutations). For instance, AFL [1] retains the mutants that can be executed to increase edge coverage as seeds for further mutations. Albeit coverage-guided fuzzing strategies have been shown effective [25], [31]–[33], they are also argued to be deficient in exploring hard-to-cover program states (e.g., certain conditional jumps between basic blocks) [4], [5], [9], [24], [34].

B. Concolic Execution

We first introduce symbolic execution [35]–[38] which inputs programs with symbolic variables and tracks program execution via an interpreter (e.g., KLEE [36]) or instrumentation (e.g., SymCC [39]) to represent all runtime variable states as symbolic expressions. Moreover, by leveraging the power of its inclusive SMT solver [30], symbolic executors can automatically generate test inputs to solve specific program path constraints. Accordingly, concolic execution [17], [40], [41] refers to tracing a **conc**rete execution path and performs the symb**olic** execution simultaneously. Specifically, one widely-adopted type of concolic executors negates each

conditional branch in an execution path under concrete input for solving the corresponding path constraints, such as [17], [38], [39], [42], [43]. When being applied in fuzzing, concolic execution can be advanced in exploring hard-to-cover program transitions such that their solutions can be used as seeds to facilitate further exploration of program states via fuzzing. For instance, Figure 1 presents an execution path with two conditional statements (i.e., ψ_1 and ψ_2) in the upper right corner. Correspondingly, a concolic executor solves the negated path constraints $\neg\psi_1$ and $\psi_1 \wedge \neg\psi_2$ respectively such that the scope bounded by ψ_1 and ψ_2 can be further explored. While compared with coverage-guided fuzzing strategies which are usually lightweight in promptly exploring program states, concolic execution tends to be heavyweight, i.e., incurring significant computation overhead for symbolic emulation and constraint solving [10], [22], [23].

C. Coordination Mode

As mentioned, in this paper, we focus on hybrid fuzzers which develop a coordination mode to coordinate the usage of its fuzzing strategy and concolic executor. Figure 1 shows that a coordination mode typically includes two components, i.e, the scheduling and synchronization mechanisms, which are illustrated as follows.

1) Scheduling: Scheduling refers to selecting and sorting the subjects for performing concolic execution. In general, the existing scheduling mechanisms are mainly seed-oriented. For instance, many hybrid fuzzers [10], [12], [13], [24] randomly select seeds for concolic execution. Moreover, DigFuzz [14] prioritizes seeds according to their quantitative difficulty of exploring edges, and MEUZZ [15] adopts a machine learning-based regression model to predict the seed utility for seed scheduling. Note that for a hybrid fuzzer, an ideal scheduling mechanism is expected to fully leverage the power of the fuzzing strategy and the concolic execution for their respective purposes rather than mixing their usage to cause redundant program state exploration.

2) Synchronization: Synchronization refers to the manner of inputting the solutions, i.e., the resulting mutants, of concolic execution as the seeds for activating the execution of fuzzing strategies. Essentially, synchronization can advance the fuzzing strategy to further explore new program states bounded by the solved path constraints [10], [24]. In general, most existing hybrid fuzzers simply iteratively execute their fuzzing strategies upon the termination of their concolic executions. Considering that the mutations in fuzzing strategies can easily invalidate the path constraints solved by concolic execution and thus compromise the exploration on program states, Pangolin [16] converts the path constraint to the polyhedral abstraction domain [44] with the SMT-opt algorithm [45] for limiting mutation space. Accordingly, Pangolin adopts a sampling-based algorithm named Dikin walk [46] to uniformly sample the polyhedral abstraction to generate the mutants.

In this paper, we study the performance and rationale of not only the hybrid fuzzers, but also their technical components.

TABLE I
STUDIED HYBRID FUZZERS

Name	Conference	Fuzzing Strategy	Concolic Executor	Coordination Mode	
				Sch†	Sync‡
QSYM [10]	USENIX Security'18	AFL	QSYM-ce	R	D
Angora [11]	S&P'18	AFL	Angora-ce	R	D
Eclipser [12]	ICSE'19	AFL	Eclipser-ce	R	D
Intriguer [13]	CCS'19	AFL	Intriguer-ce	R	D
DigFuzz [14]	NDSS'19	AFL	QSYM-ce	MC	D
MEUZZ [15]	RAID'20	AFL	QSYM-ce	ML	D
Pangolin [16]	S&P'20	AFL	QSYM-ce	R	PD

†**Scheduling** - R: Random, MC: Monte Carlo, ML: Machine Learning
‡**Synchronization** - D: Default, PD: Polyhedral Path Abstraction + Dikin Walk

III. EXTENSIVE STUDY

A. Subjects and Benchmarks

1) Subject: To determine our study subjects, we first select the hybrid fuzzers recently published in prestigious software engineering and system security conferences, e.g., ICSE, FSE, CCS, S&P, and USENIX Security. Next, we filter the selected hybrid fuzzers based on the availability of their source code and the feasibility of their execution environments.

Eventually, we select seven hybrid fuzzers for study as in Table I. We can observe that they all adopt AFL [1] as their fuzzing strategy. In particular, QSYM [10], Angora [11], Eclipser [12], and Intriguer [13] propose their specific concolic executor designs. On the other hand, DigFuzz [14], MEUZZ [15], and Pangolin [16] attempt to strengthen their coordination modes. We present their details as follows.

QSYM [10], proposed as one baseline hybrid fuzzer, tailors a concolic executor with fast symbolic emulation and enhanced constraint solving strategy.

Angora [11] attempts to replace the concolic executor of QSYM by approximating its path constraint solver with taint tracking and gradient descent search. Note that although Angora is not a typical hybrid fuzzer, we still include it as a baseline for our study as it shares the same insight as other hybrid fuzzers.

Eclipser [12] applies a grey-box concolic executor which leverages lightweight instrumentation to infer and solve approximated branch conditions.

Intriguer [13] attempts to address the constraint solving issues at the field level, i.e., using field inference and field transition tree to simplify symbolic emulation. Meanwhile, Intriguer adopts the SMT solver only for complicated constraints.

DigFuzz [14] schedules seeds by modeling the difficulty of exploring edges for each seed as a probability using the Monte Carlo method [47] and prioritizes the seeds by ranking their probabilities for concolic execution.

MEUZZ [15] adopts a linear regression model to predict the seed utility based on feature engineering and data labeling, for seed scheduling.

Pangolin [16] improves the synchronization mechanism by formulating the path constraint as polyhedral path abstraction and adopting Dikin walk [46] to sample the mutants as input seeds for its fuzzing strategy.

To comprehensively evaluate our study subjects, we further include typical conventional coverage-guided fuzzers (AFL [1], FairFuzz [48], and AFL++ [49]) for performance comparison. Since all the studied hybrid fuzzers adopt one core for fuzzing strategy and concolic execution respectively, in this paper, we implement the two-instance versions of the conventional coverage-guided fuzzers following [50] for fair performance comparison. In particular, we simply replicate their original single-core (instance) fuzzing strategies in an additional core (instance) and run them simultaneously. The two instances also perform synchronization periodically via sharing their individually generated seeds.

2) Benchmark Programs: Following prior studies [25], [27], [51], we construct our benchmark with the commonly adopted programs in the studied hybrid fuzzers [10]–[16]. As a result, we collect 15 real-world programs with their latest versions shown in Table II to evaluate code coverage and bug detection for our study.

TABLE II
STUDIED REAL-WORLD BENCHMARK

Program	Version	Input format	Argument
readelf	binutils-2.37	ELF	-a @@
nm	binutils-2.37	ELF	-C @@
objdump	binutils-2.37	ELF	-D @@
strip	binutils-2.37	ELF	@@
tcpdump	commit-465a8f	PCAP	-r @@
libxml2	2.9.12	XML	@@
libjpeg	v9c	JPEG	@@
jhead	commit-f0a884	JPEG	@@
libpng	1.7.0	PNG	@@
libtiff	4.2.0	TIFF	@@
file	commit-d17d8e	FILE	-m magic @@
bento	commit-7ddec0	MP4	@@
wavpack	commit-36b08d	WAV	-y @@
cyclonedds	commit-53cf7c	IDL	@@
libming	commit-04aee5	SWF	@@

B. Experiment Setup

Prior study [28] indicates that inappropriate seed choices can lead to high variance in evaluation results and thus potentially cause untenable performance. To alleviate such issue, we strictly follow the instructions in the previous work [16], [27], [28], [52] to construct the initial seed corpora for reflecting the real-world testing scenarios and reducing the bias of the edge coverage results. In particular, for the benchmark programs whose input formats are JPEG, PNG, and TIFF as in Table II, we collect their corresponding AFL seed collection [53]. For the rest programs, we adopt the seed collection from their original projects. Then we employ `afl-cmin` to eliminate duplicate files to minimize the corpora size. Note that we keep our initial seed corpora identical across all experimental runs.

We adopt edge coverage to represent code coverage, as our studied hybrid fuzzers [10]–[16]. Here an edge represents a conditional jump between two basic blocks in programs. All the evaluation results are averaged for 5 experimental runs for reducing the impact caused by randomness. Following prior work [5], [11], [14], [16], [27], the execution time budget for each fuzzer is set to be 24 hours for all our experiments.

All the experiments are conducted on ESC servers with 2.6 GHz AMD EPYC™ ROME 7H12 CPUs and 256 GiB RAM running Linux 4.15.0-147-generic Ubuntu 18.04.

C. Research Questions

We investigate the following research questions for extensively studying hybrid fuzzing:

- **RQ1:** *How do hybrid fuzzers perform on top of our benchmark programs?* For this RQ, we evaluate the performance of the studied hybrid fuzzers under multiple setups.
- **RQ2:** *How do existing coordination modes impact hybrid fuzzers?* For this RQ, we investigate the performance impact of the existing coordination modes.

D. Results and Analysis

1) RQ1: Performance of hybrid fuzzers: Table III demonstrates the edge coverage results of the studied fuzzers upon our benchmark. Surprisingly, we observe that the baseline technique QSYM achieves the optimal performance on average (5,763 edges), followed by Pangolin (5,561 edges) and Angora (5,517 edges). Moreover, QSYM dominates in 7 out of 15 studied benchmark programs. Specifically, while DigFuzz and MEUZZ are respectively equipped with Monte Carlo method [47] and supervised regression model [54] to strengthen the scheduling mechanism of QSYM, they nevertheless underperform QSYM by 7.67% and 5.92%. Similarly, while Pangolin attempts to improve the synchronization mechanism of QSYM, it underperforms QSYM by 3.51%.

We further attempt to analyse the edge coverage comparison results presented in the original papers of the studied hybrid fuzzers. Note that Angora, Eclipser and DigFuzz only compare their results with conventional coverage-guided fuzzers (e.g., AFL) while failing to include any hybrid fuzzer. Thus, we only include Intriguer, MEUZZ and Pangolin for analysis. In particular, we analyse the commonly studied benchmark programs between our study and their original papers (i.e., 4 for Intriguer, 6 for MEUZZ, and 9 for Pangolin). Surprisingly, while all the original papers demonstrate edge coverage improvement over QSYM (12.42% for Intriguer, 6.60% for MEUZZ and 21.90% for Pangolin), QSYM outperforms Intriguer, MEUZZ, and Pangolin by 3.75%, 9.99%, and 0.17% respectively in our study.

Interestingly, we observe that different papers present rather inconsistent edge coverage results on the same projects, e.g., for program *readelf*, QSYM explores 6,012, 1,244, 8,402, and 9,512 edges respectively in the original Intriguer, MEUZZ, Pangolin papers, and our study. Since all evaluations follow the same setups, e.g., metric and execution time, we infer that the performance variances are caused by the divergent hardware platforms and initial seed corpora applied in different papers.

Finding 1: The edge coverage comparison results among hybrid fuzzers from their original papers may not well generalize to other experimental setups.

TABLE III
EDGE COVERAGE RESULTS OF THE STUDIED FUZZERS

Program	AFL	FairFuzz	AFL++	QSYM	Angora	Eclipser	Intriguer	DigFuzz	MEUZZ	Pangolin
readelf	9,176	9,198	9,154	9,512	9,632	9,220	9,620	9,378	9,487	**10,053**
nm	5,127	5,258	5,216	5,602	6,255	5,327	5,154	5,209	5,907	**7,082**
objdump	7,358	7,317	7,415	**8,304**	7,356	7,455	7,743	7,285	7,203	7,894
strip	6,340	7,104	6,788	7,624	7,582	7,210	6,906	7,541	7,195	**7,940**
tcpdump	9,782	9,879	9,955	**10,279**	10,025	10,170	9,302	10,018	9,502	9,320
libxml2	5,876	5,860	5,929	**7,888**	5,909	5,935	5,844	5,945	5,958	5,797
libjpeg	2,902	2,806	2,981	**3,183**	3,101	3,169	2,988	3,120	3,147	3,168
jhead	304	304	304	**885**	304	823	796	747	812	304
libpng	1,496	1,517	1,503	2,058	**2,170**	1,523	1,466	1,485	2,083	1,915
libtiff	3,546	3,642	3,508	3,793	**3,883**	3,764	3,710	3,704	3,761	3,697
file	2,283	2,327	2,346	2,553	**2,571**	2,148	2,342	2,466	2,331	2,391
bento	3,001	3,020	3,135	4,017	3,937	3,566	3,495	3,356	3,855	**4,119**
wavpack	5,703	5,721	5,683	5,797	5,756	5,780	5,612	5,745	5,633	**5,803**
cyclonedds	4,822	4,871	5,012	**5,612**	5,260	4,914	4,885	5,017	5,402	4,956
libming	8,197	8,742	8,775	**9,335**	9,021	8,847	8,941	8,794	9,048	8,983
AVG	5,061	5,171	5,180	5,763	5,517	5,323	5,254	5,321	5,422	5,561

The optimal result in the conventional coverage-guided fuzzers is highlighted in blue and the corresponding inferior results in the hybrid fuzzers are highlighted in green.
The optimal result for each program among all the studied fuzzers is marked in **red**.

Note that although hybrid fuzzers are proposed to enhance conventional coverage-guided fuzzers by injecting concolic execution [10], [22]–[24], [55], limited research effort has been made to indicate their exact performance advantages. In particular, multiple studied hybrid fuzzers only compare their performance with AFL, while Pangolin and MEUZZ compare with AFLFast [2] additionally. Thus, we then investigate the edge coverage comparison between the hybrid fuzzers and conventional coverage-guided fuzzers, as shown in Table III where the overall optimal results, the optimal results achieved by the conventional coverage-guided fuzzers, and the corresponding inferior results achieved by the hybrid fuzzers are marked. We observe that averagely, all the hybrid fuzzers are more effective than the conventional coverage-guided fuzzers, e.g., the worst-performing hybrid fuzzer Intriguer still slightly outperforms the top-performing conventional coverage-guided fuzzer AFL++ (5,254 edges vs. 5,180 edges). However, such performance advantages are compromised compared with their original papers. For instance, Angora and Eclipser outperform AFL by 9.01% and 5.18% respectively in our study while by 27.08% and 25.15% respectively in their original papers. Moreover, while all our studied hybrid fuzzers outperform AFL in terms of edge coverage upon each individual collected benchmark program in their original papers, such comparison results are also somewhat refuted in our study. For instance, AFL can outperform Intriguer on 4 out of 15 programs and MEUZZ on 3 out of 15 programs.

> *Finding 2: The edge coverage advantages of hybrid fuzzers over conventional coverage-guided fuzzers are somewhat limited, indicating that the power of concolic execution has not been fully unleashed.*

Inspired by Findings 1 and 2, we further investigate the performance impact from the fuzzing strategies and concolic executors. To this end, we establish a group of hybrid fuzzer variants by reassembling their fuzzing strategies and concolic executors while retaining their original coordination modes. Specifically, we select the three studied conventional coverage-guided fuzzers as the fuzzing strategy options. For the concolic executors (represented as the "hybrid fuzzer name-ce"), since DigFuzz, MEUZZ and Pangolin all adopt QSYM-ce as their concolic executors, we select QSYM-ce, Angora-ce, Eclipser-ce, and Intriguer-ce as the concolic executor options. The average edge coverage results of our studied hybrid fuzzer variants are shown in Figure 2 where each column represents the edge coverage result of one hybrid fuzzer variant combining one fuzzing strategy and one concolic executor. Note that the columns labeled with "Origin" refer to the edge coverage results of the conventional coverage-guided fuzzers. We then observe that simply changing fuzzing strategies or concolic executors alone incurs limited impact on the edge coverage results. For instance, combining QSYM-ce with different fuzzing strategies (i.e., AFL, FairFuzz, and AFL++) results in 5,763, 5,830 and 5,842 explored edges respectively where the maximum performance gap is merely 79 edges. Moreover, when combining AFL++ with different concolic executors, the optimal edge coverage result is achieved by combining with QSYM-ce (5,842 explored edges) even though Eclipser and Intriguer are proposed to improve over QSYM.

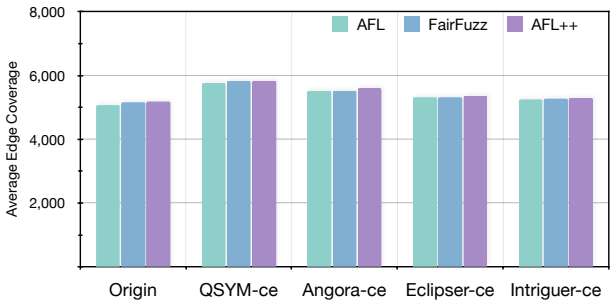

Fig. 2. Edge coverage results of reassembled hybrid fuzzing variants

Finding 3: Simply updating fuzzing strategies or concolic executors alone in hybrid fuzzers leads to limited edge coverage impact.

TABLE IV
UNIQUE CRASHES ON REAL-WORLD BENCHMARK

Program	AFL	FairFuzz	AFL++	QSYM	Angora	Eclipser	Intriguer	DigFuzz	MEUZZ	Pangolin
readelf	5	6	5	6	7	5	7	4	4	8
nm	10	10	11	10	13	12	7	9	9	13
objdump	3	3	3	3	3	2	3	0	0	3
tcpdump	6	6	5	7	6	4	8	4	3	4
libjpeg	36	30	30	34	28	22	26	18	21	34
jhead	0	2	5	16	0	15	15	12	16	0
libtiff	0	2	3	2	0	1	1	0	0	2
bento	11	19	20	25	28	15	15	21	20	16
cyclonedds	28	30	36	34	32	25	27	29	42	37
libming	10	8	10	10	12	6	10	8	8	10
Total	109	116	128	147	129	107	119	105	123	127

We further evaluate the studied fuzzers in terms of the unique crashes which are derived by analyzing the coverage updates upon program crashes following prior work [2], [3], [11], [16], [48], [56]. Table IV presents the results of unique crashes which occur on 10 of 15 benchmark programs. We can observe that while QSYM can achieve the non-negligible advantage over the conventional coverage-guided fuzzers, e.g., exposing 19 more unique crashes than AFL++, the rest hybrid fuzzers incur somewhat limited or even no advantages. For instance, Intriguer and MEUZZ only expose 3 and 7 more crashes respectively than FairFuzz. Additionally, AFL exposes more unique crashes than Eclipser and DigFuzz (109 vs. 107 and 105 crashes) and dominates in project *libjpeg* (36 crashes).

Finding 4: Most studied hybrid fuzzers incur rather limited or even no advantages over conventional coverage-guided fuzzers in exposing unique crashes upon real-world benchmark programs.

2) RQ2: Impact of coordination mode: Our earlier findings altogether indicate that the power of hybrid fuzzers has been compromised so far and updating either their fuzzing strategies or concolic executors alone leads to limited effect. Accordingly, we then investigate how their coordination modes impact fuzzing performance. Note that fuzzing strategies are leveraged for prompt program state exploration and concolic executors are leveraged for exploring hard-to-cover program branches [10], [14], [16], [22], [24]. Ideally, they should explore no common edges. Thus, to evaluate the effectiveness of the coordination modes adopted by our studied hybrid fuzzers, we propose a metric namely *redundant edge ratio* to reflect the magnitude of the common edges explored by the fuzzing strategy and the concolic executor of a hybrid fuzzer. In particular, we calculate the *redundant edge ratio* ϕ via Equation 1.

$$\phi(F, C) = \frac{|F \cap C|}{|C|} \quad (1)$$

where C and F refer to the total edges firstly explored by executing the concolic executor and the fuzzing strategy (i.e., AFL) respectively. Ideally, their intersection should be empty. However, due to separate global coverage states, fuzzing strategy synchronizes coverage states periodically from concolic executor, causing inevitable coverage-updating gaps. Thus, fuzzing strategy and concolic executor can repeatedly explore the same edges, causing non-empty intersection between C and F. Intuitively, the larger *redundant edge ratio* is, the more common program branches are solved by the concolic executor and the fuzzing strategy, compromising the effectiveness of the concolic executor, i.e., degrading the effectiveness of hybrid fuzzers.

Figure 3 presents the *redundant edge ratio* results of our studied hybrid fuzzers for each benchmark program. Surprisingly, we observe quite significant average *redundant edge ratio*s for most of the benchmarks, ranging from 0.47 (*jhead*) to 0.95 (*libjpeg*). For instance, in project *libjpeg*, the *redundant edge ratio* is larger than 0.95 in all the studied hybrid fuzzers except QSYM (0.80). Such results indicate that our studied hybrid fuzzers incur quite severe redundant edge exploration between fuzzing strategies and concolic executors. Moreover, combining with Table III, we further find that the performance of hybrid fuzzing is highly correlated with *redundant edge ratio*. In particular, QSYM achieves the optimal edge coverage results (5,763 edges) and the lowest *redundant edge ratio* (0.65) among all. Meanwhile, Eclipser and Intriguer respectively achieve inferior edge coverage (5,323 and 5,254 edges) and higher *redundant edge ratio*s (0.92 and 0.91). Moreover, for project *jhead*, the edge coverage advantage of hybrid fuzzers is rather significant (i.e., 300+ explored edges for all the conventional coverage-guided fuzzers vs. 885 and 823 explored edges respectively for QSYM and Eclipser). Note that the *redundant edge ratio* in project *jhead* is even 0 for QSYM and MEUZZ. We then examine the source code of project

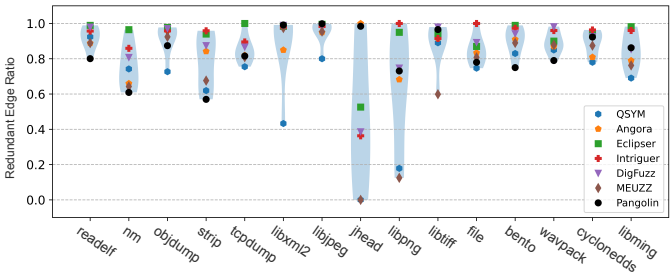

Fig. 3. *Redundant edge ratio* of studied fuzzers

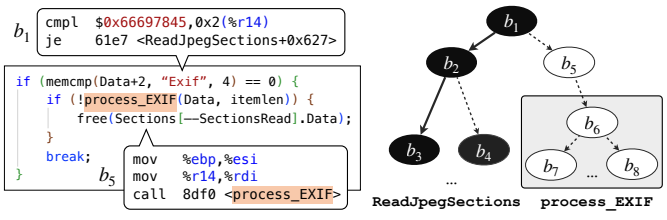

Fig. 4. An example in project *jhead*

jhead to infer the cause. In particular, Figure 4 presents a code snippet of project *jhead* which calibrates the *if* condition to find out whether string "Exif" (i.e., 0x66697845 in its binary form) is matched in function `ReadJpegSection` (the complete code and its corresponding labels are presented in [21] due to page limit). For conventional coverage-guided fuzzers, the probability of generating the feasible input matching such 32-bit value via probabilistic mutation is $1/2^{32} \approx 2.3 \times 10^{-10}$. Therefore, we infer that the chance to explore the edge associated with such *if* condition is rather low, i.e., the edge exploration of the scope under such *if* condition can be easily halted. However, applying concolic execution can quickly solve the path constraint of the *if* condition such that fuzzing strategies are further leveraged to promptly explore the edges within function `process_EXIF`. Such results enlighten that for a hybrid fuzzer, its coordination mode potentially plays a vital role for impacting its effectiveness.

Finding 5: The hybrid fuzzing effectiveness is reflected by redundant edge ratio which is highly relevant to their coordination modes.

We further investigate the performance impact from the individual components of the coordination mode. First, we investigate how the scheduling mechanism affects the edge coverage. As mentioned, DigFuzz and MEUZZ propose their scheduling mechanisms with the Monte Carlo method [47] and the supervised regression model [54] respectively in order to schedule the seeds according to their capabilities for exploring hard-to-cover edges, while the rest studied hybrid fuzzers randomly select seeds, for concolic execution. However, we find from Figure 3 that QSYM outperforms DigFuzz in all of the programs and MEUZZ in 10 out of 15 programs where the average *redundant edge ratio* is 0.65 for QSYM, 0.71 for MEUZZ and 0.87 for DigFuzz. Such results indicate that the seed scheduling mechanisms adopted by our studied hybrid fuzzers do not effectively limit *redundant edge ratio*.

Finding 6: The seed scheduling mechanisms adopted by our studied hybrid fuzzers do not effectively alleviate the issue of redundant edge exploration.

We also attempt to investigate the effect of synchronization. Note that only Pangolin specifically designs its synchronization mechanism while the rest studied hybrid fuzzers simply input the resulting seeds from the concolic execution to the fuzzing strategy. Specifically, Pangolin first linearizes path constraint to the polyhedra abstraction domain with the SMT-opt algorithm [45] for limiting mutation spaces. Then Pangolin utilizes a sampling algorithm, i.e., Dikin walk [46] to sample the abstraction domain to derive the mutants as the seeds for the fuzzing strategy. We then concentrate our analysis on comparing the edge coverage results between QSYM and Pangolin because they only differ in their adopted synchronization mechanisms. Surprisingly, although Pangolin attempts to enhance QSYM via its specifically designed synchronization mechanism, it still performs worse in terms of edge coverage, i.e., 5,561 edges vs. 5,763 edges on average. Such result indicates that the existing effort on strengthening the synchronization mechanism of hybrid fuzzers may pose rather limited performance impact.

Finding 7: The existing effort on strengthening the synchronization mechanism may have limited impact on the edge coverage performance of hybrid fuzzers.

E. Discussion

Previous findings indicate that while the coordination modes significantly impact the performance of hybrid fuzzers, the existing effort on the scheduling and synchronization mechanisms lead to rather limited effectiveness. We then discuss the possible reasons. In particular, we first discuss why enhancing seed scheduling in the studied hybrid fuzzers (i.e. DigFuzz and MEUZZ) does not effectively alleviate the redundant edge exploration (Finding 6). More specifically, during concolic execution, each conditional branch along the execution path of PUT is negated to solve the corresponding path constraints. Note that such effort can be cost-ineffective upon the conditional branches which can be explored by fuzzing strategies usually with much lower overhead. Moreover, the coverage updates on applying fuzzing strategies and concolic executions are mutually unknown for all our studied hybrid fuzzers. Therefore, the concolic executor is likely to spend massive effort on solving the edges which have been explored by the fuzzing strategy. Typically, fine-grained scheduling is expected to successfully rule out the redundant edge exploration. Unfortunately, even when MEUZZ and DigFuzz attempt to shepherd their seed scheduling mechanisms via refined machine learning or statistics approaches, they fail to distinguish the worth of each edge to be explored by concolic executors, and thus are ineffective on preventing redundant edge exploration. Accordingly, we infer that proposing finer-grained edge-oriented rather than seed-oriented scheduling mechanisms is essential.

We then discuss why the synchronization mechanism of Pangolin causes limited performance impact. Specifically, we infer that the polyhedra abstraction domain [44] is not an optimal choice out of the three sound abstract domains (interval [19], octagon [57] and polyhedra) used for limiting mutation space with different precision levels and time cost. Specifically, the polyhedra abstraction domain is adopted by Pangolin due to its highest precision for approximating path constraints which results in the fewest false positives (i.e., mutants which cannot be executed to increase code coverage). However, the fuzzing strategy adopted by Pangolin, i.e., AFL, can quickly filter out such mutants [1], [24], [25] such that adopting any abstract domain actually leads to close effects on limiting the mutation space. Thus, one could totally adopt the interval abstraction domain with the lowest time cost instead.

In addition to Dikin walk [46] adopted by Pangolin, there are also other representative sampling algorithms (e.g., Hit-and-run [58], Vaidya walk [20] and John walk [20]) for the

$\mathbb{R}^d$ domain defined by n constraints where n is the total number of path constraints with d corresponding symbolic inputs (generally $n \gg d$) for concolic execution in our paper. Notably, compared with the time complexity of Dikin walk ($O(n^2d^3)$), the time complexity of John walk ($O(nd^{4.5})$) is less dependent on n, i.e., the amount of path constraints, while more dependent on d, i.e., the amount of program symbolic inputs. Since in real-world programs, the amount of path constraints largely exceeds the amount of program symbolic inputs [30], [45], applying John walk thus can be more efficient, i.e., exploring more edges under given time.

IV. Enhancing Hybrid Fuzzers

A. Approach

Inspired by our previous findings and discussion, we propose CoFuzz (**Co**ordinated hybrid **fuzz**ing framework with advanced coordination mode) which strengthens the hybrid fuzzer performance by optimizing their scheduling and synchronization mechanisms on top of QSYM. Figure 5 presents the workflow of CoFuzz. Specifically, CoFuzz first applies the *edge-oriented scheduling* which identifies the edges with unexplored sibling edges (marked as ❶). Then, CoFuzz extracts the corresponding branch features (❷) as input to an online linear regression model [59] based on Stochastic Gradient Descent (SGD) [18] for predicting the utility and scheduling the edges for concolic executors (❸). Note that we select the online linear regression model because the model expects that the overall coefficients can be estimated under diverse benchmark programs where the coefficients typically converge fast. Moreover, by taking the resulting seeds after applying the concolic executors (❹), CoFuzz applies the *sampling-augmenting synchronization* mechanism which adopts the interval abstraction domain with John walk to generate mutants as input for fuzzing strategies (❺). Meanwhile, the *sampling-augmenting synchronization* mechanism incrementally updates our online linear regression model via the edge utility represented as the corresponding coverage updates (❻). The details of CoFuzz are illustrated as follows.

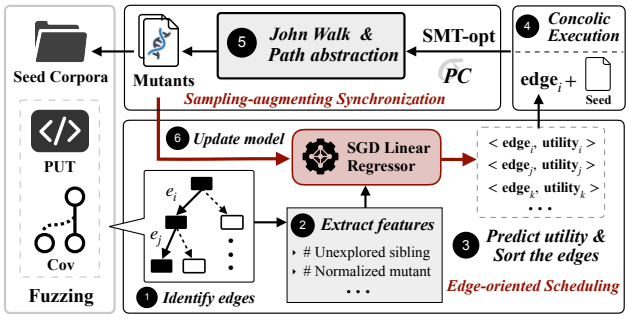

Fig. 5. The framework of CoFuzz

1) Edge-oriented Scheduling: To essentially alleviate the redundant edge exploration between fuzzing strategies and concolic executors, we propose the fine-grained *edge-oriented scheduling* mechanism to schedule edges corresponding to the branches in program execution path for applying the concolic executors. Algorithm 1 presents the details of the *edge-oriented scheduling*, we first identify the edges having at least one unexplored sibling edge, i.e., the edge under one sharing prefix edge, by analyzing runtime code updates (line 3). Next, we predict the utility, i.e., the coverage updates, for these edges to be solved by the concolic executor using the linear regression model with SGD (line 4). More specifically, inspired by the existing work [14], [15], [48], [60], [61], we extract the following five features covering the general program and mutant features and the challenges of accessing conditional branches. Accordingly, we form a five-dimensional vector as the input of the online linear regression model of our *edge-oriented scheduling*.

Algorithm 1: Coordination Mode of CoFuzz

```
   Input: Model
   Result: res
1  Function PerformingCoordinationMode:
2      res ← set()
3      candidates ← Set of edges with unexplored sibling edges
4      utility ← Model.predict(candidates) ;   ▷ predict using
         the linear regression model with SGD
5      critical_edges ← edgeSchedule(candidates, utility) ;
         ▷ schedule the edges with high utility
6      for all edge e_i in critical_edges do
7          s_i ← Identify the seed covering e_i
8          pc ← concolicExec(s_i, e_i)
9          φ̂ ← SMTopt(pc)
10         sample_set ← JohnWalk(s_i, φ̂)
11         for mutant in sample_set do
12             if increaseCoverage(mutant) then
13                 res.add(mutant)
14         end
15         cov_i ← Increased coverage
16         Model.update(e_i, cov_i)
17     end
18     return res
```

1. ***Edge distance to the root.*** As in [15], [61], this feature refers to the shortest distance traversing from the given edge to the root to reflect the potential of generating solutions by the concolic executor [61].
2. ***Count of unexplored sibling edges.*** This feature reflects the potential of exploring edges as in [15], [26]. For a given edge, the larger such count is, the more possible that more program states, e.g., the uncovered cases when executing the switch statements, can be explored.
3. ***Normalized mutant amount.*** For a given edge, we find first all its explored sibling edges and then the fuzzer-generated mutants exploring them. Finally, we collect the mutant amount and normalize it with log transformation. This metric reflects the effort by the fuzzing strategy to explore the given edge, as in [14], [48].
4. ***Conditional branch type.*** This feature includes equality predicates (e.g. `cmp` instruments with `eq` or `neq`) and statements containing comparison functions (e.g., `memcmp` and `strcmp`) to reflect the challenges of generating the

feasible input via random mutation, as in [15], [62].

5. ***Condition bit width.*** This feature refers to the bit width of operands in a condition as [62], [63] to reflect the challenge of covering such statement via random mutation.

Then, we schedule the edges with high utility (i.e., namely *critical edges* in this paper) and leverage the concolic executor to solve them (lines 5 to 6). More specifically, we identify the seed whose execution path covers the *critical edge* (line 7) to activate the concolic executor to only negate such a *critical edge* for solving the corresponding constraint (line 8). In this way, we significantly alleviate the redundant edge exploration, i.e., such *critical edges* can only be solved by concolic executors exclusively instead of fuzzing strategies. For instance, by adopting *edge-oriented scheduling* to the sample code in Figure 4, we only schedule edge (b_1, b_2) for concolic execution since edge (b_2, b_3) has no unexplored sibling edge. Then the concolic executor negates the condition in b_1 and solves the path constraint to explore the *critical edge* (b_1, b_5), preventing the redundant exploration of edge (b_2, b_4).

2) Sampling-augmenting Synchronization: Algorithm 1 (lines 8 to 10) demonstrates our *sampling-augmenting synchronization* mechanism which consists of three steps: (1) generating the path constraint pc via concolic execution for the target *critical edge* from *edge-oriented scheduling*, (2) converting pc to the interval path abstraction $\hat{\varphi}$ by the SMT-opt algorithm [45] (i.e., determining the scale of each corresponding input), and (3) sampling in the abstraction domain with John walk and generating the mutants. Accordingly, we filter the mutants failing to explore new edges and store the resulting seeds for future executions of fuzzing strategies (line 13). Meanwhile, we collect the coverage updates as the utility for each solved edge, and update the online SGD regressor, i.e., performing incremental learning (lines 15 to 16). Note that the *sampling-augmenting synchronization* mechanism is proposed to not only increase the edge coverage by inputting the mutants efficiently sampled within limited mutation space for the fuzzing strategy, but also advance *edge-oriented scheduling* by utilizing the edge increase to update the online linear regression model simultaneously, i.e., enhancing the prediction accuracy of the edge utility during *edge-oriented scheduling*.

B. Evaluation

We evaluate the performance of CoFuzz in terms of edge coverage and bug detection with the identical evaluation setups as in Section III-B where the results are shown as follows.

1) Edge Coverage: We select AFL and the top-performing hybrid fuzzer QSYM for performance comparison. To perform ablation study on our *edge-oriented scheduling* and *sampling-augmenting synchronization* mechanisms respectively, we build the corresponding CoFuzz variants, i.e., CoFuzz$_{sch}$ with *edge-oriented scheduling* only and CoFuzz$_{sync}$ with *sampling-augmenting synchronization* only. Table V presents the edge coverage results for our study subjects. We can observe that overall, CoFuzz outperforms AFL and QSYM by 32.44% and 16.31% respectively in terms of edge coverage and dominates all the benchmark programs. We further perform significance tests to investigate the robustness of the edge coverage advantage of CoFuzz over QSYM. Following prior work [16], [25], [27], [52], we leverage Mann Whitney U-test [64] with one-tailed hypothesis to measure the significance of such performance advantage. We calculate the p-value between the edge coverage performance of QSYM and CoFuzz for each program in terms of two significance levels (i.e., 0.01 and 0.05). We can observe that the p-values listed in Table V are smaller than 0.01 for 13 of 15 programs and way smaller than 0.05 for the rest 2 programs, i.e., *libjpeg* and *libtiff* (both 0.01059). Such results indicate that CoFuzz can significantly and consistently dominate all our studied hybrid fuzzers.

We further observe that CoFuzz$_{sch}$ and CoFuzz$_{sync}$ can outperform all the studied hybrid fuzzers, e.g., outperforming QSYM by 11.05% and 7.37% respectively. Note that CoFuzz outperforms CoFuzz$_{sch}$ and CoFuzz$_{sync}$ by 4.73% and 8.32% which indicates that jointly improving the scheduling and synchronization mechanisms is rather essential to optimize the overall performance of hybrid fuzzers. Such results also indicate that *edge-oriented scheduling* and *sampling-augmenting synchronization* can potentially boost each other. Additionally, we evaluate the temporal development of CoFuzz in terms of edge coverage. Figure 6 presents the average edge coverage for all benchmark programs of CoFuzz and other studied fuzzers over time. Overall, we can observe that CoFuzz achieves better performance compared with other studied fuzzers. Specifically, the edge coverage of CoFuzz is significantly increased after each synchronization stage, indicating the effectiveness of concolic execution. Note that we also present the results for all benchmark programs which show the similar trends in our GitHub repository [21] due to page limit.

TABLE V
EDGE COVERAGE RESULTS OF COFUZZ

Program	AFL	QSYM	CoFuzz$_{sch}$	CoFuzz$_{sync}$	CoFuzz	p-value
readelf	9,176	9,512	10,407	10,236	10,786	0.00596
nm	5,127	5,602	7,824	7,573	8,234	0.00609
objdump	7,358	8,304	8,512	8,453	8,710	0.00609
strip	6,340	7,624	7,839	8,598	9,094	0.00609
tcpdump	9,782	10,279	12,661	10,348	13,130	0.00609
libxml2	5,876	7,888	8,493	7,946	8,640	0.00609
libjpeg	2,902	3,183	3,192	3,190	3,210	0.01059
jhead	304	885	897	890	915	0.00199
libpng	1,496	2,058	2,239	2,197	2,311	0.00609
libtiff	3,546	3,793	3,820	3,842	3,974	0.01059
file	2,283	2,553	2,652	2,730	2,851	0.00609
bento	3,001	4,017	5,624	5,398	6,179	0.00609
wavpack	5,703	5,797	5,832	5,857	5,863	0.00596
cyclonedds	4,822	5,612	5,832	5,713	5,932	0.00609
libming	8,197	9,335	10,177	9,846	10,719	0.00609
AVG	5,061	5,763	6,400	6,188	6,703	0.00640

2) Bug Detection: Following all our studied hybrid fuzzers [10]–[16], we evaluate CoFuzz on the LAVA-M dataset with the time budget of 5 hours. Note that while LAVA-M automatically injects and labels bugs into four programs (*base64*, *md5sum*, *uniq*, *who* in *coreutils-8.24*), it is also quite common for fuzzers to detect more bugs than the listed ones [11], [12], [16]. Table VI presents the number of detected bugs N and the corresponding bug survival time T_m in minutes by all the studied techniques. We can observe that they all

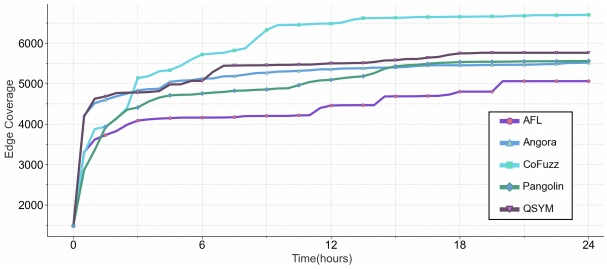

Fig. 6. Average edge coverage of CoFuzz and other studied fuzzers over time

fully expose the listed bugs in the subjects *base64*, *md5sum* and *uniq* while CoFuzz spends way less time than others, i.e., around 66 seconds for CoFuzz vs. 604 seconds for the second fastest hybrid fuzzer Angora. Meanwhile, for program *who*, although none of the hybrid fuzzers full expose the injected bugs within 5 hours, CoFuzz still exposes the most bugs (1,913 bugs) which outperforms Angora by 23.66%.

We further evaluate CoFuzz in terms of unique crashes on real-world benchmark programs. Overall, CoFuzz exposes a total of 456 unique crashes (around 2X more than other hybrid fuzzers combining with Table IV). Additionally, CoFuzz exposes the crashes in three programs (*strip*, *file* and *wavpack*) which cannot be detected by any other studied hybrid fuzzer. The detailed crash information is presented in our Github repository [21] due to the page limit. We further manually calibrate all the crashed files to derive 42 bugs including 37 previously unknown bugs all of which are only detected by CoFuzz. Table VII presents the bug details where 30 previously unknown bugs have been confirmed by the developers with 8 new CVEs and 20 of them have been fixed. Note that for the 5 patched bugs in the latest version, CoFuzz can still detect different execution paths to trigger them.

TABLE VI
BUG RESULTS OF COFUZZ ON LAVA-M

Fuzzer	base64		md5sum		uniq		who	
	N	T_m	N	T_m	N	T_m	N	T_m
QSYM	44/44	8.48	57/57	31.77	28/28	4.55	1,332/2,136	300.00
Angora	48/44	6.75	57/57	16.37	29/28	7.15	1,547/2,136	300.00
Eclipser	46/44	128.33	57/57	147.35	29/28	155.83	1,030/2,136	300.00
Intriguer	46/44	205.07	57/57	132.60	29/28	187.22	1,350/2,136	300.00
DigFuzz	46/44	7.53	57/57	57.30	28/28	4.32	1,146/2,136	300.00
MEUZZ	44/44	7.28	57/57	40.35	28/28	6.50	1,205/2,136	300.00
Pangolin	48/44	9.37	57/57	132.75	29/28	13.27	1,342/2,136	300.00
CoFuzz	48/44	1.07	57/57	1.75	29/28	0.50	1,913/2,136	300.00

V. THREATS TO VALIDITY

Threats to internal validity. One threat to internal validity lies in the implementation of the studied subjects. To reduce this threat, we reuse the source code of the studied hybrid fuzzers and their runtime environment as we can. For DigFuzz and Pangolin without publicly available source code, we strictly follow the description in their papers for re-implementation where the first three authors carefully review our code to ensure the correctness and consistency. Another threat to internal validity lies in the reliability of our evaluation results which can possibly be compromised by randomness. Accordingly, all our results are averaged from five runs, following prior work [11], [14], [15], [27], [62], [65].

Threats to external validity. The threat to external validity mainly lies in the subjects and benchmarks. To reduce this threat, we select 7 representative hybrid fuzzers recently published in prestigious software engineering and system security conferences as mentioned in Section III-A1.We also apply 15 real-world benchmark programs which are frequently used in the original papers of our studied hybrid fuzzers and the LAVA-M dataset in our evaluation.

Threats to construct validity. The threat to construct validity mainly lies in the adopted metrics in our study. To reduce this threat, we follow many existing fuzzers [5], [16], [25], [27], [48], [66] to adopt the edge coverage as our evaluation metric. Moreover, we also evaluate the bug detection capability of the studied hybrid fuzzers on top of the LAVA-M benchmark and our real-world benchmark programs.

TABLE VII
BUGS DETECTED BY COFUZZ IN REAL-WORLD BENCHMARK

Program	Function	Bug Type	Count	Bug Status
readelf	process_object	memory leaks	1	Confirmed
nm	demangle_path	stack-buffer-overflow	1	Confirmed
	str_buf_append	stack-buffer-overflow	1	Patched
objdump	unknow module	invalid memory reference	1	Patched
strip	bfd_getl32	heap-buffer-overflow	3	CVE-2022-38533 & Fixed
	bfd_getl32	invalid memory reference	2	Patched
	group_signature	heap-use-after-free	1	Patched
libjpeg	jpeg_read_scanlines	use-of-uninitialized-value	1	Confirmed
jhead	ReadJpegSections	use-of-uninitialized-value	1	CVE-2022-37165
libtiff	_tiffMapProc	use-of-uninitialized-value	2	Reported
	tiffcp	heap-buffer-overflow	1	Confirmed & Fixed
file	file_tryelf	allocation-size-too-big	1	Confirmed & Fixed
bento	ParseExtension	heap-buffer-overflow	1	CVE-2022-37167 & Fixed
	WriteBytes	heap-buffer-overflow	1	CVE-2022-37169
	AP4_HvccAtom	heap-buffer-overflow	3	CVE-2022-37690
	AP4_StsdAtom	invalid memory reference	2	CVE-2022-37166 & Fixed
	AP4_AvccAtom	invalid memory reference	1	CVE-2022-37168 & Fixed
	Create	memory leaks	2	CVE-2022-37691
wavpack	MD5_Final	heap-buffer-overflow	2	Confirmed & Fixed
cyclonedds	parse_line	heap-buffer-overflow	3	Confirmed & Fixed
	idl_reference_node	heap-use-after-free	2	Confirmed & Fixed
	idlc_parse	stack-buffer-overflow	2	Confirmed & Fixed
	idl_parse	invalid memory reference	2	Confirmed & Fixed
libming	newVar_N	heap-buffer-overflow	2	Reported
	decompileAction	invalid memory reference	3	Reported

VI. RELATED WORK

A. Fuzzing

Many fuzzers adopt coverage to guide fuzzing. AFL [1] retains the mutants executed to increase code coverage as seeds. Accordingly, AFLFast [2] leverages the Markov chain model [67] to prioritize low-frequency execution paths, and AFLGo [3] introduced directed fuzzing which generates inputs for the target program. FairFuzz [48] increases the edge coverage of AFL by facilitating the exploration of rare branches identified at runtime. Mopt [56] utilizes the particle swarm optimization algorithm to schedule mutators. EnFuzz [68] integrates different fuzzing techniques with a synchronization mechanism. Additionally, fuzzing has been widely applied in domain-specific software systems. SGFUZZ [69] is proposed

to test stateful software systems like network protocol implementations. Deephunter [70] fuzzes deep neural networks with extensible coverage criteria. Zhang et al. [71] combine GANs and metamorphic testing to generate image mutants for fuzzing autonomous driving systems. Zhou et al. [72] further paint the generated image mutants on billboards to enhance their real-world applicability. Liu et al. [73] propose FANS to detect vulnerabilities in Android native system services by generating test cases for specific interfaces. Wu et al. [74] fuzz CUDA programs by building memory models and the rules to detect memory conflicts for exposing their synchronization bugs. They also use the fuzzing results to guide the process of fixing them accordingly [75]. More recently, Zhao et al. [76] propose JavaTailor to learn information from historical bug-revealing test programs for generating test programs to expose JVM defects, while Wu et al. [77] further propose JITFuzz which leverages multiple mutators to fully exercise the JIT optimization components for fuzzing JVM JIT compilers.

Many fuzzers are proposed to enhance coverage-guided fuzzing strategies when exploring hard-to-cover program states. VUzzer [78] integrates its evolutionary fuzzing strategy via dynamic taint analysis. Steelix [34] leverages light-weight program analysis to acquire program state information and locate the offsets for magic bytes. Greyone [65] infers taints of variables driven by fuzzing and utilizes them to guide program state exploration. Aschermann et al. [4] exploit input-to-state relations to improve fuzzing effectiveness. Specifically, hybrid fuzzers leverage the power of concolic executors in addition to fuzzing strategies. QSYM [10] tailors a concolic executor with fast symbolic emulation and enhanced constraint solving. Accordingly, Angora [11] adopts gradient descent to solve path constraints. Eclipser [12] proposes grey-box concolic testing to resolve conditional branches. Intriguer [13] proposes field-level constraint solving which applies the SMT solver only for complicated conditions. On the other hand, DigFuzz [14] proposes probabilistic seed prioritization with the Monte Carlo method. MEUZZ [15] utilizes a machine learning regression model to predict seed utility for seed scheduling. The more recent Pangolin [16] uses Dikin walk to uniformly sample a polyhedron path abstraction. In this paper, we propose CoFuzz to enhance the coordination mode of hybrid fuzzers for augmenting the overall fuzzing effectiveness.

Researchers also tend to study the existing fuzzers to guide the future relevant research. Klees et al. [27] investigate the experiment setup and statistical analysis methods for reliable fuzzing evaluation. Liang et al. [29] present the major obstacles and their solutions while applying fuzzing in practice. Boehme et al. [8] investigate challenges and opportunities for fuzzing and symbolic execution. Herrera et al. [28] study how to construct initial seed corpora for fuzzing. Moreover, some researchers study the rationale of fuzzing strategies. Wang et al. [79] investigate and evaluate the efficacy of machine learning techniques in the existing fuzzers. Ding et al. [80] investigate characteristics and life cycles of the detected faults of OSS-Fuzz, a continuous fuzzing service for open source software. Wu et al. [25] evaluate a generic stochastic mutation strategy adopted in many existing fuzzers and improve the strategy with a reinforcement learning model. They also find that while combining deep learning and program smoothing can be helpful for fuzzing, it still can be improved by including a mechanism for identifying edge properties [26]. In this paper, we conduct the first extensive study on hybrid fuzzers and demonstrate that their coordination modes significantly impact the overall performance for the first time.

B. Symbolic/Concolic Execution

While symbolic execution has been applied in vulnerability exploitation for long time [30], [35], it tends to incur high computation overhead and path explosion. To tackle such issues, KLEE [36] reserves the solved constraint states to eliminate the repetitive computation costs. Trabish et al. [81] propose chopped symbolic execution, which alleviates path explosion by targeting important code fragments and leverages static analysis to resolve side effects. Li et al. [82] propose the heuristic to guide symbolic execution to insufficient explored program states. Concolic execution [17] combines symbolic execution and concrete execution [40], [83] to improve efficiency and scalability in large softwares. SAGE [84] performs concolic execution at the binary level and mitigates the problem of scalability. Triton [43] is a concolic execution framework which allows dynamic binary analysis. Poeplau et al. [39] propose SymCC which compiles the concolic execution right in the binary level to facilitate the performance. Very recently, Liu et al. [85], [86] have also applied symbolic/concolic constraint solving to test the emerging machine learning systems.

VII. Conclusion

In this paper, we have extensively investigated hybrid fuzzers. Specifically, we first find for many studied hybrid fuzzes, their performance may not well generalize to other experimental setups. We further find that their edge coverage performance is highly related to the redundant edge exploration between applying fuzzing strategy and concolic execution. Inspired by our findings, we propose CoFuzz with *edge-oriented scheduling* and *sampling-augmenting synchronization*. The evaluation results demonstrate that CoFuzz can significantly outperform QSYM by 16.31% in terms of edge coverage and expose 2X more unique crashes than all studied hybrid fuzzers with 37 previously unknown bugs detected.

VIII. Acknowledgement

This work is partially supported by the National Natural Science Foundation of China (Grant No. 61902169), Guangdong Provincial Key Laboratory (Grant No. 2020B121201001), and Shenzhen Peacock Plan (Grant No. KQTD2016112514355531). This work is also partially supported by National Science Foundation under Grant Nos. CCF-2131943 and CCF-2141474, as well as Ant Group. Yuqun Zhang would like to dedicate this paper to the memory of his grandmother. She will live in his heart forever.

REFERENCES

[1] M. Zalewski, "American fuzz lop," https://github.com/google/AFL, 2020.

[2] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[3] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[4] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence." in *NDSS*, vol. 19, 2019, pp. 1–15.

[5] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program smoothing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 803–817.

[6] J. Liu, Y. Wei, S. Yang, Y. Deng, and L. Zhang, "Coverage-guided tensor compiler fuzzing with joint ir-pass mutation," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, apr 2022. [Online]. Available: https://doi.org/10.1145/3527317

[7] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018.

[8] M. Boehme, C. Cadar, and A. Roychoudhury, "Fuzzing: Challenges and reflections." *IEEE Softw.*, vol. 38, no. 3, pp. 79–86, 2021.

[9] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1597–1612.

[10] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 745–761.

[11] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.

[12] J. Choi, J. Jang, C. Han, and S. K. Cha, "Grey-box concolic testing on binary code," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 736–747.

[13] M. Cho, S. Kim, and T. Kwon, "Intriguer: Field-level constraint solving for hybrid fuzzing," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 515–530.

[14] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing." in *NDSS*, 2019.

[15] Y. Chen, M. Ahmadi, B. Wang, L. Lu *et al.*, "{MEUZZ}: Smart seed scheduling for hybrid fuzzing," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 77–92.

[16] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang, "Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1613–1627.

[17] K. Sen, "Concolic testing," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007, pp. 571–572.

[18] L. Bottou, "Stochastic gradient descent tricks," in *Neural networks: Tricks of the trade*. Springer, 2012, pp. 421–436.

[19] P. Cousot *et al.*, "Static determination of dynamic properties of programs." 1977.

[20] Y. Chen, R. Dwivedi, M. J. Wainwright, and B. Yu, "Fast mcmc sampling algorithms on polytopes," *The Journal of Machine Learning Research*, vol. 19, no. 1, pp. 2146–2231, 2018.

[21] G. Repository, "Hybrid fuzzing approach," https://github.com/Tricker-z/CoFuzz, 2022.

[22] R. Majumdar and K. Sen, "Hybrid concolic testing," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 416–426.

[23] B. S. Pak, "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution," *School of Computer Science Carnegie Mellon University*, 2012.

[24] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, no. 2016, 2016, pp. 1–16.

[25] M. Wu, L. Jiang, J. Xiang, Y. Huang, H. Cui, L. Zhang, and Y. Zhang, "One fuzzing strategy to rule them all," in *Proceedings of the International Conference on Software Engineering*, 2022.

[26] M. Wu, L. Jiang, J. Xiang, Y. Zhang, G. Yang, H. Ma, S. Nie, S. Wu, H. Cui, and L. Zhang, "Evaluating and improving neural program-smoothing-based fuzzing," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 847–858. [Online]. Available: https://doi.org/10.1145/3510003.3510089

[27] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[28] A. Herrera, H. Gunadi, S. Magrath, M. Norrish, M. Payer, and A. L. Hosking, "Seed selection for successful fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 230–243.

[29] J. Liang, M. Wang, Y. Chen, Y. Jiang, and R. Zhang, "Fuzz testing in practice: Obstacles and solutions," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 562–566.

[30] L. d. Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[31] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "Fuzzbench: an open fuzzer benchmarking platform and service," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1393–1403.

[32] J. Song and J. Alves-Foss, "The darpa cyber grand challenge: A competitor's perspective," *IEEE Security & Privacy*, vol. 13, no. 6, pp. 72–76, 2015.

[33] K. Serebryany, "{OSS-Fuzz}-google's continuous fuzzing service for open source software," 2017.

[34] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 627–637.

[35] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[36] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[37] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 1066–1071.

[38] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[39] S. Poeplau and A. Francillon, "Symbolic execution with {SymCC}: Don't interpret, compile!" in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 181–198.

[40] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.

[41] X. Wang, J. Sun, Z. Chen, P. Zhang, J. Wang, and Y. Lin, "Towards optimal concolic testing," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 291–302.

[42] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[43] F. Saudel and J. Salwan, "Triton: A dynamic symbolic execution framework," in *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes*, 2015, pp. 31–54.

[44] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1978, pp. 84–96.

[45] Y. Li, A. Albarghouthi, Z. Kincaid, A. Gurfinkel, and M. Chechik, "Symbolic optimization with smt solvers," *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 607–618, 2014.

[46] R. Kannan and H. Narayanan, "Random walks on polytopes and an affine interior point method for linear programming," *Mathematics of Operations Research*, vol. 37, no. 1, pp. 1–20, 2012.

[47] J. Hammersley, *Monte carlo methods*. Springer Science & Business Media, 2013.

[48] C. Lemieux and K. 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, 2018, pp. 475–485.

[49] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "{AFL++}: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[50] M. Zalewski, "Afl parallel fuzzing," https://github.com/google/AFL/blob/master/docs/parallel_fuzzing.txt, 2020.

[51] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng *et al.*, "Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers." in *USENIX Security Symposium*, 2021, pp. 2777–2794.

[52] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, "Savior: Towards bug-driven hybrid testing," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1580–1596.

[53] M. Zalewski, "Afl official seed corpus," http://lcamtuf.coredump.cx/afl/demo/, 2021.

[54] L. Bottou *et al.*, "Online learning and stochastic approximations," *Online learning in neural networks*, vol. 17, no. 9, p. 142, 1998.

[55] M. Wang, J. Liang, Y. Chen, Y. Jiang, X. Jiao, H. Liu, X. Zhao, and J. Sun, "Safl: increasing and accelerating testing coverage with symbolic execution and guided fuzzing," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018, pp. 61–64.

[56] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "{MOPT}: Optimized mutation scheduling for fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1949–1966.

[57] A. Miné, "The octagon abstract domain," *Higher-order and symbolic computation*, vol. 19, no. 1, pp. 31–100, 2006.

[58] L. Lovász and S. Vempala, "Hit-and-run from a corner," *SIAM Journal on Computing*, vol. 35, no. 4, pp. 985–1005, 2006.

[59] S. Weisberg, *Applied linear regression*. John Wiley & Sons, 2005, vol. 528.

[60] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Model-based whitebox fuzzing for program binaries," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 543–553.

[61] S. Cha, S. Hong, J. Lee, and H. Oh, "Automatically generating search heuristics for concolic testing," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 1244–1254.

[62] J. Liang, M. Wang, C. Zhou, Z. Wu, Y. Jiang, J. Liu, Z. Liu, and J. Sun, "Pata: Fuzzing with path aware taint analysis," in *2022 2022 IEEE Symposium on Security and Privacy (SP)(SP). IEEE Computer Society, Los Alamitos, CA, USA*, 2022, pp. 154–170.

[63] A. Fioraldi, D. C. D'Elia, and E. Coppa, "Weizz: Automatic grey-box fuzzing for structured binary formats," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 1–13.

[64] P. E. McKnight and J. Najab, "Mann-whitney u test," *The Corsini encyclopedia of psychology*, pp. 1–1, 2010.

[65] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "{GREYONE}: Data flow sensitive fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2577–2594.

[66] D. She, R. Krishna, L. Yan, S. Jana, and B. Ray, "Mtfuzz: fuzzing with a multi-task neural network," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 737–749.

[67] J. R. Norris and J. R. Norris, *Markov chains*. Cambridge university press, 1998, no. 2.

[68] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "{EnFuzz}: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1967–1983.

[69] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, "Stateful greybox fuzzing," *arXiv preprint arXiv:2204.02545*, 2022.

[70] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See, "Deephunter: a coverage-guided fuzz testing framework for deep neural networks," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 146–157.

[71] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 132–142. [Online]. Available: https://doi.org/10.1145/3238147.3238187

[72] H. Zhou, W. Li, Z. Kong, J. Guo, Y. Zhang, B. Yu, L. Zhang, and C. Liu, "Deepbillboard: Systematic physical-world testing of autonomous driving systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 347–358. [Online]. Available: https://doi.org/10.1145/3377811.3380422

[73] B. Liu, C. Zhang, G. Gong, Y. Zeng, H. Ruan, and J. Zhuge, "{FANS}: Fuzzing android native system services via automated interface analysis," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 307–323.

[74] M. Wu, Y. Ouyang, H. Zhou, L. Zhang, C. Liu, and Y. Zhang, "Simulee: Detecting cuda synchronization bugs via memory-access modeling," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 937–948. [Online]. Available: https://doi.org/10.1145/3377811.3380358

[75] M. Wu, L. Zhang, C. Liu, S. H. Tan, and Y. Zhang, "Automating cuda synchronization via program transformation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 748–759.

[76] Y. Zhao, Z. Wang, J. Chen, M. Liu, M. Wu, Y. Zhang, and L. Zhang, "History-driven test program synthesis for jvm testing," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1133–1144. [Online]. Available: https://doi.org/10.1145/3510003.3510059

[77] M. Wu, M. Lu, H. Cui, J. Chen, Y. Zhang, and L. Zhang, "Jitfuzz: Coverage-guided fuzzing for jvm just-in-time compilers," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. New York, NY, USA: Association for Computing Machinery, 2023.

[78] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing." in *NDSS*, vol. 17, 2017, pp. 1–14.

[79] Y. Wang, P. Jia, L. Liu, C. Huang, and Z. Liu, "A systematic review of fuzzing based on machine learning techniques," *PloS one*, vol. 15, no. 8, p. e0237749, 2020.

[80] Z. Y. Ding and C. Le Goues, "An empirical study of oss-fuzz bugs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 131–142.

[81] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, "Chopped symbolic execution," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 350–360.

[82] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," *ACM SigPlan Notices*, vol. 48, no. 10, pp. 19–32, 2013.

[83] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 213–223.

[84] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[85] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, "Nnsmith: Generating diverse and valid test cases for deep learning compilers," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 530–543. [Online]. Available: https://doi.org/10.1145/3575693.3575707

[86] J. Liu, J. Peng, Y. Wang, and L. Zhang, "Neuri: Diversifying dnn generation via inductive rule inference," *arXiv preprint arXiv:2302.02261*, 2023.