Tumbling Down the Rabbit Hole: How do Assisting Exploration Strategies Facilitate Grey-box Fuzzing?

Mingyuan Wu[†‡]
Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology
Shenzhen, China
11849319@mail.sustech.edu.cn

Jiahong Xiang[†‡]
Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology
Shenzhen, China
xiangjh2022@mail.sustech.edu.cn

Kunqiu Chen
Southern University of Science and Technology
Shenzhen, China
11911626@mail.sustech.edu.cn

Peng Di
Ant Group
Hangzhou, China
dipeng.dp@antgroup.com

Shin Hwei Tan
Concordia University
Montreal, Canada
shinhwei.tan@concordia.ca

Heming Cui
The University of Hong Kong
Hong Kong, China
heming@cs.hku.hk

Yuqun Zhang[†*]
Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology
Shenzhen, China
zhangyq@sustech.edu.cn

***Abstract*—Many assisting exploration strategies have been proposed to assist grey-box fuzzers in exploring program states guarded by tight and complex branch conditions such as equality constraints. Although they have shown promising results in their original papers, their evaluations seldom follow equivalent protocols, e.g., they are rarely evaluated on identical benchmarks. Moreover, there is a lack of sufficient investigations on the specifics of the program states explored by these strategies which can obfuscate the future application and development of such strategies. Consequently, there is a pressing need for a comprehensive study of assisting exploration strategies on their effectiveness, versatility, and limitations to enlighten their future development. To this end, we perform the first comprehensive study about the assisting exploration strategies for grey-box fuzzers. Specifically, we first collect nine recent fuzzers representing the mainstream assisting exploration strategies as our studied subjects and 21 real-world projects to form our benchmark suite. After evaluating the subjects on the benchmark suite, we then surprisingly find that the dictionary strategy is most promising since it not only achieves similar or even slightly better performance over the other studied assisting exploration strategies in terms of exploring program states but also is more practical to be enhanced. Accordingly, we propose CDFuzz, which generates a customized dictionary for each seed upon the baseline fuzzer AFL to improve over the original dictionary strategy. The evaluation results demonstrate that CDFuzz increases the edge coverage by 16.1% on average for all benchmark projects over the best performer in our study (i.e., AFL++ with the dictionary strategy). CDFuzz also successfully exposed 37 previously unknown bugs, with nine confirmed and seven fixed by the corresponding developers.**

I. Introduction

Fuzzing has been widely adopted to expose the vulnerabilities of software systems by producing invalid, unexpected, or random data as test inputs [1]. Particularly, given a collection of seeds, grey-box fuzzers [2], [3], [4], [5] iteratively mutate them for generating new seeds to optimize their exploration on program states, i.e., executed code guarded by branch conditions [6], via obtaining real-time coverage feedback based on instrumenting target programs.

Although many grey-box fuzzers are effective in exploring sufficient program states to expose software vulnerabilities, they are still ineffective in exploring certain program states (e.g., the ones guarded by *equality constraint*s [7], [8], [9], [5]) unless developers design specific mutators for their own applications [10], [11]. To improve program state exploration, researchers have proposed multiple assisting exploration strategies which are usually implemented as an independent phase in an existing fuzzer (e.g., implementing an SMT solver [12] as a constraint-solving phase into AFL [2]) to assist general grey-box fuzzers [13], [14], [15], [16] for exploring such program states. To our best knowledge, there are four types of assisting exploration strategies: (1) the dictionary strategy, which enables a list of tokens that fuzzers can insert to mutants to cope with the grammar-blind problem (i.e., generating input that may violate grammar) [14], (2) the input-to-state correspondence strategy, which utilizes lightweight taint tracking to monitor how input values are used at various states during program execution. In this way, the correspondences between the operands of a given instruction and the given input can be derived to first identify the offsets via lightweight taint tracking and then update their values for exploring the associated program states [15], [17], (3) the SMT-solver-based strategy, which leverages SMT solvers [12] to solve constraints that satisfy complex branch conditions for exploring program states (e.g., QSYM [18]), and (4) the gradient-based strategy, which utilizes gradient descent [19] to solve constraints for exploring program states (e.g., Angora [16]). Although these strategies have been shown effective in their corresponding

[*]Yuqun Zhang is the corresponding author.
[‡]These authors contributed equally.
[†]These authors are also affiliated with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China. Mingyuan Wu is also affiliated with the University of Hong Kong, Hong Kong, China.

papers, their evaluation can be potentially biased since they seldom follow equivalent protocols, e.g., while Angora and QSYM both use eight real-world projects in their original evaluations, they only adopt two projects (*objdump* and *file*) in common. Moreover, the evaluations of these strategies focus mainly on the performance of the grey-box fuzzers integrating the assisting exploration strategies as a whole while neglecting the individual contributions of the strategies, e.g., the specifics of their explored program states. Without a thorough understanding of the explored program states by these strategies, it is unclear how to further improve the effectiveness of these strategies to assist grey-box fuzzers in exploring deeper program states.

In this paper, we perform the first comprehensive study to evaluate the assisting exploration strategies. Specifically, we first collect nine recent fuzzers as our studied subjects and construct a benchmark suite which consists of 21 open-source real-world projects commonly studied by their original papers. Then, we conduct an extensive evaluation where our evaluation results suggest that the dictionary strategy achieves quite similar and even slightly better performance over other studied assisting exploration strategies, e.g., AFL activating the dictionary strategy slightly outperforms the best-performing SMT-solver-based fuzzer QSYM in terms of the average edge coverage (5,094 vs. 5,067 explored edges). We observe that it is also more practical to be enhanced by strategically selecting tokens to form a dictionary for each seed.

Inspired by our study, we propose CDFUZZ (***C**ustomized **D**ictionary **Fuzz**ing*), which customizes the dictionary by strategically selecting tokens for each seed upon the baseline fuzzer AFL [2]. Specifically, CDFUZZ derives the execution path of each seed and extracts all its constant tokens in *equality constraint*s to generate a customized dictionary. Accordingly, each of such tokens is inserted to a random seed offset to generate a mutant for the further fuzzing campaign. The evaluation results indicate that under 24-hour evaluation, CDFUZZ can outperform the best performer in our study (i.e., AFL++ [3] activating the dictionary strategy) by 16.1% in terms of edge coverage. CDFUZZ also exposed 37 previously unknown bugs where 30 of them can only be exposed by CDFUZZ in our evaluation (i.e., other evaluated fuzzers can only expose 7 of them within the given time limit). Specifically, nine of them have been confirmed and seven have been fixed by the corresponding developers. To summarize, this paper makes the following contributions:

- To the best of our knowledge, we conduct the first comprehensive study of nine representative fuzzers on the performance impact of assisting exploration strategies on top of a collection of real-world projects.
- Our study revealed that the dictionary strategy is most promising because it not only achieves similar or even slightly better performance over other studied strategies, but also is more practical to be enhanced.
- We propose a lightweight approach CDFUZZ which customizes the dictionary for each seed by strategically selecting constant tokens. It outperforms the best performer in our study by 16.1% in terms of edge coverage and exposes 37 previously unknown bugs with nine confirmed and seven fixed by the corresponding developers. All experimental results and our tool is publicly available in our *GitHub* repository [20].

II. BACKGROUND

A. Grey-box Fuzzing

Grey-box fuzzing [21] has become the major practice for fuzzing. The objective of a grey-box fuzzer is to iteratively explore the target program thoroughly for exposing potential vulnerabilities. We take the widely-used baseline fuzzer American Fuzzy Lop (AFL) [2] to illustrate generic grey-box fuzzers. First, AFL instruments the coverage tracking instructions into the target program in compilation time for collecting coverage information. Next, it mutates seeds (i.e., inputs of the target program) and executes the resulting mutants on the instrumented program to obtain coverage information. If such mutants explore new program states (i.e., increase code coverage), AFL identifies such mutants as "interesting" seeds and retains them for further exploration. The procedure above is iterated throughout the entire fuzzing campaign. While grey-box fuzzers can in general advance the exploration of program states upon the explored program states progressively, they have also been shown somewhat limited in exploring the program states guarded by tight and complex branch conditions (e.g., *equality constraint*s) [18].

B. Assisting Exploration Strategies

Many assisting exploration strategies have been proposed to assist grey-box fuzzers on exploring program states guarded by tight and complex branch conditions [18] [16] [22] [15]. The general workflow of integrating assisting exploration strategies in grey-box fuzzers is shown in Figure 1 and illustrated as follows:

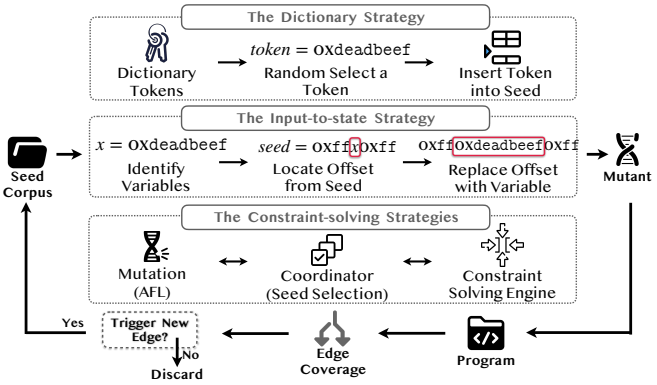

Fig. 1: The workflow of assisting exploration strategies

1) The dictionary strategy: The dictionary strategy either adopts a set of user-defined tokens (so-called "dictionary") or generates tokens automatically by parsing the constant values defined in the target program [23]. Such tokens are randomly inserted into random seed offsets to generate mutants for the further fuzzing campaign, expecting that certain tokens can be

inserted in the correct branch conditions of the target program to advance the program state exploration. For example in Figure 1, to explore the *equality constraint* `x == 0xdeadbeef`, the token `0xdeadbeef` in the dictionary can be either given by users or parsed from code. Next, the token `0xdeadbeef` is inserted to a random seed offset to generate a mutant. If the token happens to be inserted in the offset of the input corresponding to the righthand side of the *equality constraint* `x == 0xdeadbeef`, i.e., the *equality constraint* is satisfied by executing the mutant, its guarded program state can be explored.

2) The input-to-state correspondence strategy: Since it is likely that partial input can be stored in the memory or registers at run time, the input-to-state correspondence strategy derives the correspondence between such input and the associated program state to facilitate fuzzing. Specifically, the input-to-state correspondence strategy identifies the lefthand and righthand side of a branch condition during runtime via additional instrumentation. Next, it identifies which side originated from the corresponding seed, and then locates its offset via lightweight taint tracking. Furthermore, it updates such offset with the value of the other side, i.e., generating a mutant to be executed for satisfying the corresponding branch condition such that its guarded program state can be explored. For instance in Figure 1, given a branch condition `x == 0xdeadbeef`, the input-to-state correspondence strategy first identifies the value of `x`, and then traces the offset of `x` in the seed. At last, `x` is updated with the value `0xdeadbeef` in the seed to generate a new mutant which is then input to the target program for exploring the program states guarded by `x == 0xdeadbeef`.

3) The constraint-solving strategies (the SMT-solver-based strategy and the gradient-based strategy): Symbolic execution [24] is widely used in grey-box fuzzers by leveraging full taint tracking and constraint solving to generate seeds satisfying target branch conditions. In particular, the SMT-solver-based strategy utilizes SMT-solver [12] and the gradient-based strategy utilizes gradient descent [19] for solving constraints, respectively. To illustrate, in Figure 1, the grey-box fuzzer and the constraint-solving engine (i.e., the symbolic/concolic execution engine or the gradient descent solver) are first activated at the same time. Next, the grey-box fuzzer passes its generated seeds to the constraint-solving engine which leverages full taint tracking to derive constraints corresponding to the seed inputs and solve them for exploring program states guarded by tight and complex branch conditions. Meanwhile, the constraint-solving engine also passes its generated seeds to the grey-box fuzzer for the further fuzzing campaign [13].

Although all these strategies have been well evaluated in their original papers, their evaluations can be potentially biased because (1) they seldom follow identical protocols and (2) the assessment of their individual contributions is rather obscure. Thus, there is a pressing need to comprehensively study the assisting exploration strategies to enlighten their future development.

III. Empirical Study

A. Subjects & Benchmarks

1) Subjects: We aim at the grey-box fuzzers with assisting exploration strategies as our study subjects. In particular, we filter many such fuzzers for selecting the representative ones. Following prior studies [13], [25], we first limit our search scope to the fuzzers recently published in the top Software Engineering and Security conferences (e.g., ICSE, FSE, CCS, and S&P). Furthermore, we can only evaluate the fuzzers which are publicly available and can be successfully executed. Lastly, as it is rather challenging and time-consuming to activate certain strategies, e.g., the dictionary strategy, in non-AFL-based fuzzers, we only target AFL-based fuzzers.

Finally, we select nine representative fuzzers as our studied subjects. Specifically, AFL [2], AFL++ [3], MOPT [4], and FAIRFUZZ [5] represent the dictionary-based fuzzers (while they deactivate the dictionary strategy option by default, it can be easily activated as long as the associated tokens are provided). QSYM [18], MEUZZ [26], and PANGOLIN [22] represent the SMT-solver-based fuzzers. Angora [16] and REDQUEEN [15] are typical gradient-based fuzzer and input-to-state-correspondence-based fuzzer, respectively. Note that REDQUEEN is implemented into AFL++ (which is maintained by Google) as the REDQUEEN mode [27] and thus we choose it as the representative fuzzer of input-to-state correspondence strategy following prior work [28].

2) Benchmark suite: Following prior work [13], we construct our benchmark suite based on the projects commonly adopted by the original papers of the selected fuzzers [3], [4], [5], [15], [16], [18]. To ensure general applicability of our study, we additionally adopt seven projects from FuzzBench [29], resulting in a total of 21 real-world projects. In particular, we select 14 frequently used projects out of the papers to form our benchmark suite. More specifically, we first select seven projects that are adopted by at least three papers. Then, we randomly select another seven projects which are adopted by one or two papers. The selection details are presented in our *GitHub* page [20]. Table I presents the statistics of our benchmark suite. Specifically, we consider our benchmark to be sufficient and representative due to the following reasons:

1) These 21 benchmark projects cover ten different file formats for seed inputs, e.g., `ELF`, `XML`, `JPEG`, and `JSON`;
2) The sizes of these programs that range from 1,885 to over 150K lines of code (LoC) can represent a wide range of programs in practice;
3) They are all open-source real-world programs from different vendors with various code logic.
4) They cover diverse functionalities including development tools (e.g., *readelf*, *objdump*), xml processing tools (e.g., *xmlwf*), network analysis tools (e.g., *tcpdump*), graphics processing tools (e.g., *djpeg*), etc.

TABLE I: Statistics of the studied benchmarks

Programs				LOC
Package	Target	Commit/Version	Class	
binutils	readelf	2.40	ELF	72,164
	nm	2.40	ELF	55,307
	objdump	2.40	ELF	74,532
	size	2.40	ELF	54,429
	strip	2.40	ELF	65,432
libjpeg	djpeg	9c	JPEG	9,023
tcpdump	tcpdump	4.99.0	PCAP	46,892
libxml2	xmllint	2.9.12	XML	73,320
jhead	jhead	3.04	JPEG	1,885
libpng	pngfix	1.6.36	PNG	12,173
libtiff	tiffinfo	4.2.0	TIFF	15,140
expat	xmlwf	2.4.8	XML	6,871
libtiff	tiff2bw	4.2.0	TIFF	15,024
mupdf	mutool	1.18.0	PDF	123,575
libjpeg-turbo*	libjpeg-turbo	3b19db	JPEG	11,106
libpng*	libpng	cd0ea2	PNG	31,054
libxml2*	libxml2	c7260a	XML	104,019
re2*	re2	b025c6	REGEX	17,754
jsoncpp*	jsoncpp	8190e0	JSON	4,181
sqlite3*	sqlite3	c78cbf	SQL	95,815
bloaty*	bloaty	52948c	ELF	152,845

* These benchmark packages come from FuzzBench [29].

B. Environment Setup and Implementation

Our evaluation was conducted on the ESC servers with 128-core 2.6 GHz AMD EPYC™ ROME 7H12 CPUs and 256 GB RAM. The servers run on Linux 4.15.0-147-generic 64-bit Ubuntu 18.04. We strictly follow the respective original procedures of the studied fuzzers when executing them. Specifically, to allow the fuzzers to generate more tests, we set the execution time budget for all the experiments 24 hours. Meanwhile, as all fuzzers rely on randomized algorithms, we run each experiment five times to obtain the average result, following prior evaluations [28], [26], [16]. Notably, since all the studied fuzzers are AFL-based, we apply the AFL (v2.57b, which is the latest released version in GitHub) llvm-mode (LLVM-13) to instrument the source code during compilation and LLVM IR [30] for presenting and analyzing programs. At last, we collect the initial seed corpus following prior work [5], [9], [22], [31], [32].

We adopt *edge coverage* to measure code coverage where an edge refers to a transition between program blocks, e.g., a conditional jump, following prior work [6], [22]. Specifically, we compute edge coverage via the unique edge number derived by the AFL built-in tool named `afl-showmap`, which has been widely used by many existing fuzzers [4], [16], [22], [28], [33], [34].

Constructing *Dictionary* versions via FuzzingDriver. To form dictionaries for all studied fuzzers involving dictionary strategy [35], [23], we automatically extract tokens using the most recent FuzzingDriver [23] instead of relying on user-provided tokens to reduce potential bias. Specifically, FuzzingDriver is designed to automatically generate dictionaries for each program, leveraging CodeQL [36] to extract key pieces of information from the target program's internals, including commonly occurring keywords, strings, and constants. Moreover, it employs a data cleaning module that scrutinizes extracted tokens to customize the dictionary for enhancing the efficiency of the fuzzing process. [1]

C. Research Questions

We investigate the following research questions in our study:

- **RQ1:** How well do different assisting exploration strategies perform on our benchmark suite?
- **RQ2:** What are the specifics of program states explored by assisting exploration strategies?
- **RQ3:** What are the potential obstacles of different assisting exploration strategies?

D. Result Analysis

1) RQ1: Effectiveness of the studied fuzzers: Table II shows the edge coverage results of all studied fuzzers where the numbers show the coverage results averaged over multiple runs (i.e., five times). The "Orig" column denotes the original implementation of the grey-box fuzzers which deactivates the dictionary strategy and "Dict" denotes the grey-box fuzzers activating the dictionary strategy (represented as "*Fuzzer*$_{Dict}$" in this paper). In general, we can observe that all *Fuzzer*$_{Dict}$s outperform all original grey-box fuzzers in terms of the average edge coverage by 7.9% to 10.9%, e.g., AFL$_{Dict}$ explores 10.1% more edges than AFL (5,094 vs. 4,626 explored edges). Moreover, we also observe that QSYM, MEUZZ, PANGOLIN, Angora and REDQUEEN outperform the best-performing grey-box fuzzer, i.e., AFL++, by 4.7% in terms of the average edge coverage. As all the studied subjects are AFL-based, we can derive that both input-to-state correspondence strategy and constraint-solver-based strategies can somewhat advance the exploration of program states over the original grey-box fuzzers. However, we can also observe that the performance advantage is limited and may not well generalize in different benchmark projects. For instance, the best-performing constraint-solving-based fuzzer QSYM outperforms AFL++ in 11 projects by 3.5% (6,968 vs. 6,733 explored edges in *strip*) to 4.6× (731 vs. 159 explored edges in *jhead*), while AFL++ outperforms QSYM in the remaining ten projects by 1.2% (6,456 vs. 6,379 explored edges in *re2*) to 23.2% (6,178 vs. 5,013 explored edges in *sqlite3*). We also perform the Mann-Whitney U test [37] to demonstrate the significance of fuzzers that adopting assisting exploration strategies compared to the grey-box fuzzer AFL. The p-value of less than 0.05 for AFL compared to AFL$_{Dict}$, QSYM, and REDQUEEN in terms of average edge coverage indicates that fuzzers adopting assisting exploration strategies significantly outperform the grey-box fuzzer AFL.

Interestingly, we further observe from Table II that the dictionary-based fuzzers can potentially achieve similar or even slightly better performance over other studied fuzzers. For instance, AFL$_{Dict}$ outperforms QSYM by 0.5% (5,094 vs. 5,067 explored edges) on average. Since AFL$_{Dict}$ and QSYM only differ in their adopted assisting exploration strategies, it indicates that the dictionary strategy is close to or potentially

[1] Note that FuzzingDriver is not an independent fuzzer but just a tool for extracting tokens to generate dictionaries for fuzzers, we cannot adopt it as an independent baseline in our evaluation.

TABLE II: The average edge coverage result of studied programs

Benchmark	AFL		AFL++		FairFuzz		MOpt		Qsym	Meuzz	Pangolin	Angora	Redqueen
	Orig	*Dict**	*Orig*	*Dict**	*Orig*	*Dict**	*Orig*	*Dict**					
readelf	10,081	11,561	10,562	11,524	11,065	10,954	11,034	11,757	11,386	12,136	11,437	**13,203**	10,053
nm	4,973	5,199	5,651	5,270	4,877	4,830	4,966	5,580	**6,532**	5,527	6,394	5,774	5,526
objdump	5,519	5,649	5,666	5,744	5,451	**6,108**	5,518	5,760	5,999	5,832	5,593	5,863	5,587
size	3,604	3,800	4,010	4,015	3,532	3,551	3,477	3,618	**5,211**	4,151	5,045	5,201	5,002
strip	6,216	6,369	6,733	6,598	6,103	5,714	5,895	6,210	**6,968**	6,634	6,741	5,943	5,513
djpeg	2,319	2,801	2,542	2,628	2,446	2,798	2,304	2,826	2,092	2,184	2,238	**2,936**	2,463
tcpdump	10,932	12,001	11,240	12,554	11,646	**13,315**	9,409	12,051	10,053	10,117	10,843	9,502	11,471
xmllint	6,474	**6,862**	6,724	6,824	6,458	6,599	6,389	6,686	6,317	6,225	5,981	4,273	6,771
jhead	156	**801**	159	772	156	795	157	728	731	622	331	157	612
pngfix	1,123	2,237	975	1,983	998	2,023	986	2,020	1,976	**2,252**	1,903	2,161	2,189
tiffinfo	3,683	3,831	4,026	**4,209**	3,655	3,731	3,486	3,760	3,753	3,713	3,657	4,190	3,981
xmlwf	5,024	4,990	4,935	4,953	4,985	4,993	4,567	5,025	4,797	4,498	5,001	**5,132**	4,879
tiff2bw	3,112	3,503	3,198	3,478	3,531	**3,663**	3,102	3,429	2,871	3,014	3,128	3,028	3,615
mutool	2,207	2,252	2,260	2,295	2,216	2,215	2,198	2,270	2,167	2,147	2,194	2,204	**2,314**
libjpeg-turbo	4,462	4,596	4,178	**4,997**	4,900	4,905	4,997	4,997	4,722	4,641	4,785	4,772	4,554
libpng	887	2,092	884	**2,263**	1,080	2,232	1,081	2,232	2,080	1,899	2,041	1,930	2,014
libxml2	9,608	9,621	9,996	**11,267**	8,075	8,160	9,512	9,512	10,355	9,981	10,213	9,142	9,001
re2	6,458	6,465	6,456	6,466	6,419	6,464	6,391	**6,469**	6,379	6,192	6,341	6,409	6,362
jsoncpp	1,452	1,455	1,451	1,454	1,451	1,455	1,451	**1,457**	1,406	1,404	1,420	1,379	1,444
sqlite3	6,817	7,989	6,178	6,286	6,737	7,925	7,975	**8,021**	5,013	4,812	5,143	3,612	5,588
bloaty	2,043	2,909	1,912	3,514	1,912	2,989	1,980	3,052	5,595	**5,621**	5,014	5,410	5,279
Average	4,626	5,094	4,749	**5,195**	4,652	5,020	4,613	5,117	5,067	4,933	5,021	4,868	4,963
p-value	-	0.006	0.005	0.005	0.005	0.006	0.006	0.005	0.006	0.005	0.005	0.005	0.006

***Dictionaries of *Fuzzer*$_{Dict}$s are all generated by FuzzingDriver [23].**

more effective than the SMT-solver-based strategy. Such indications can be generalized when comparing the dictionary strategy with other assisting exploration strategies. Therefore, we can infer that the dictionary-based fuzzers are effective solutions in exploring program states.

Finding 1:The dictionary-based fuzzers achieve similar or even slightly better performance over other studied fuzzers, indicating that the dictionary strategy is rather effective.

2) RQ2: Specifics of the explored program states: In this paper, we characterize a *constraint* as a predicate that is represented as the edge between two basic blocks [38] in the control-flow graph (CFG). To reach a given *program state*, all its guarded constraints should be satisfied. We thus represent each program state by an ordered sequence of its guarded constraints, i.e., a sequence of branch conditions which guard the corresponding basic blocks. In particular, we consider the following two types of constraints based on LLVM IR: (1) an *equality constraint* denotes equality comparisons at the source code level (e.g., ==), which correspond to one *Jump Equal instruction* or *Jump not Equal instruction* [39] following [40]; (2) otherwise, it is referred to as a *non-equality constraint*.

Our goal is to find out which constraint in the ordered sequence of a program state is critical to be satisfied by grey-box fuzzers. Particularly, given a program state represented as the ordered sequence of its guard constraints $c_1, c_2, ..., c_k, c_{k+1}, ..., c_n$ where $c_1, ..., c_k$ are satisfiable and c_{k+1} is unsatisfiable, we consider the first unsatisfiable constraint (i.e., c_{k+1}) to be *critical* since it prevents the remaining unsatisfied constraints $(c_{k+1}, ..., c_n)$ from being explored.

We first investigate the specifics of the program states which can be explored by the fuzzers with assisting exploration strategies other than the grey-box fuzzers to reflect the improvement that the assisting exploration strategies brings to grey-box fuzzers. Specifically, we collect the unexplored program states (denoted as S) by applying the grey-box fuzzers. We then filter out the ones which cannot be explored by any fuzzer with assisting exploration strategies. Finally, we derive the *critical constraints* from the remaining program states for further analysis. We observe that the majority of the constraints which can be explored by the fuzzers integrating assisting exploration strategies but cannot be explored by their corresponding grey-box fuzzers are *equality constraint*s. Figure 2 presents the ratios of the total number of *equality constraint*s to the total number of critical constraints. For example, 98.1% of the critical constraints explored by AFL$_{Dict}$, 97.4% by AFL++$_{Dict}$, and 93.1% by Qsym, but not by their corresponding grey-box fuzzers, are equality constraints.

Finding 2: The assisting exploration strategies bring improvement over grey-box fuzzers by effectively exploring program states guarded by equality constraints.

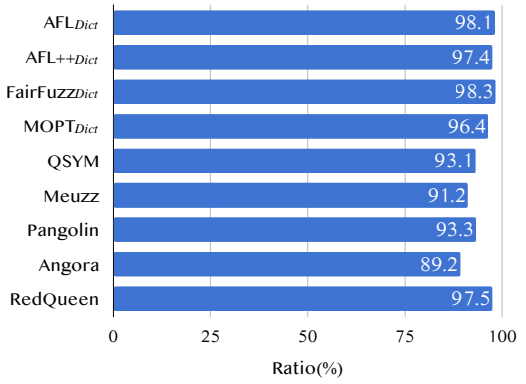

Fig. 2: Ratios of the total number of *equality constraints* to the total number of unsatisfiable critical constraints

We further investigate the characteristics for the unexplored program states by *Fuzzer*$_{Orig}$s. Note that all these program states are guarded by *equality constraint*s and can be explored by either dictionary strategy, input-to-state correspondence strategy, or constraint-solver-based strategies. Specifically, we

use a semi-automated approach to analyze the program state characteristics guarded by *equality constraint*s: (1) our script first automatically extracts *equality constraint*s, and then (2) we manually identify common characteristics among these *equality constraint*s. Surprisingly, we find that 92.3% of these unexplored *equality constraint*s share the same form as `input [==|!=] CONSTANT` (e.g., function `memcmp(input, CONSTANT)` or `switch(input){case CONSTANT}`) after compilation, i.e., taking the content directly from the input to compare with a predefined constant value (constant-evaluating *equality constraint*s).

Finding 3: The constraints explored by dictionary strategy and other studied strategies are mostly constant-evaluating equality constraints.

3) RQ3: The obstacles of different strategies: Our previous findings indicate that all assisting exploration strategies achieve similar edge coverage performance. We then discuss the potential obstacles for their future development to understand which strategy is more practical to be enhanced for better exploring the program states guarded by tight and complex branch conditions.

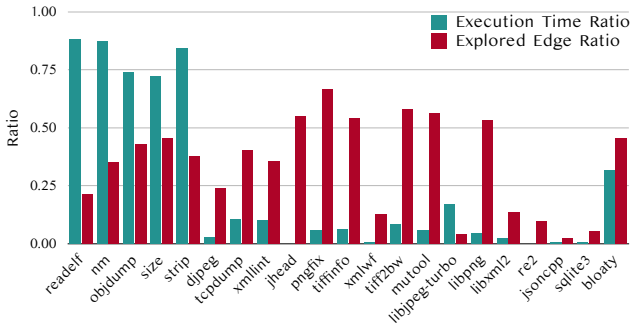

Fig. 3: Ratios of the execution time and explored edges of the input-to-state correspondence strategy

Input-to-State Correspondence Strategy. For the input-to-state correspondence strategy, we first investigate its potential effect by studying the input-to-state-correspondence-based fuzzer, i.e., REDQUEEN. Apart from input-to-state correspondence strategy, REDQUEEN also includes other strategies (e.g., the *havoc* strategy). We then investigate the ratio of the execution time of the input-to-state correspondence strategy to the overall execution time budget (i.e., 24 hours) for studying its effectiveness, as in Figure 3. We can observe considerable variations on the execution time of the input-to-state correspondence strategy across different benchmarks. For instance, in *readelf*, *nm*, and *strip*, its execution exceeds 20 hours. On the contrary, in *jhead* and *xmlwf*, its execution lasts merely less than one hour. Meanwhile, Figure 3 also presents the ratio of the edges explored by the input-to-state correspondence strategy to the overall explored edges. It is surprising to see that executing the input-to-state correspondence strategy longer does not necessarily explore more edges, e.g., executing 21 hours but only exploring 21.5% edges in *readelf*, indicating that the adopted mechanisms (e.g., lightweight taint analysis) in the input-to-state correspondence strategy can incur performance issues in certain benchmark projects. We further infer that its development may be hindered by the inaccuracies of its adopted lightweight taint tracking. To illustrate, we manually analyze 10% (271) of constant-evaluating *equality constraint*s which REDQUEEN fails to explore in our evaluation. We observe that although it successfully detected the corresponding constant values for these *equality constraint*s, its lightweight taint analysis failed to locate their corresponding offsets in the seeds. Figure 4 presents one such example from *xmllint* where REDQUEEN successfully obtains the constant value but it fails to locate its offset corresponding to `CUR_PTR` in the seed. Therefore, the *equality constraint* at line 3 cannot be satisfied, and REDQUEEN fails to explore its guarded program states.

Finding 4: The input-to-state correspondence strategy could potentially trigger performance issues in certain benchmarks while also running the risk of inaccuracies in locating correct offsets.

```
1 void xmlParseNotationDecl(xmlParserCtxtPtr ctxt) {
2   // CMP10 checks if the first 10 characters of a string 's'
        match the 10 provided characters (c1 to c10).
3   if(CMP10(CUR_PTR,'<','!','N','O','T','A','T','I','O','N'))
        {
4     // ...
5 }}
```

Fig. 4: An input-to-state fuzzing strategy failure case in *xmllint parser.c*

```
1 if (alias != XML_CHAR_ENCODING_ERROR) {
2     const char* canon;
3     // The return value of xmlGetCharEncodingName is
4     // determined by a switch statement
5     canon = xmlGetCharEncodingName(alias);
6     if ((canon != NULL) && (strcmp(name, canon))) {
7       return(xmlFindCharEncodingHandler(canon));
8 }}
9
```

Fig. 5: A case in *xmllint encoding.c*

Constraint-Solver-based Strategies. For the constraint-solver-based strategies (i.e., the SMT-solver-based strategy and the gradient-based strategy), we infer that although they can advance the exploration of program states to some extent, their effectiveness can nevertheless be compromised when exploring program states guarded by certain complex and tight branch conditions. Notably, the constraint-solver-based fuzzers fail to explore many program states guarded by *equality constraint*s which can otherwise be explored by $Fuzzer_{Dict}$s. For example in Figure 5, only the $Fuzzer_{Dict}$s can explore the condition in line 6 `(canon != NULL) && ...`) in our study. To investigate how the performance of the constraint-solver-based strategies correlates with the specifics of program states, we first characterize the "depth" of constraints. As the constraint c_{k+1} can be reached if and only if the conjunction of its all dependent constraints $c_1 \wedge c_2 \wedge ... \wedge c_k$ is satisfied, we

thus compute the "depth" for constraints c_{k+1} as the size k of the conjunction of its dependent constraints. Figure 6 demonstrates the success rate of solving constraints at different depths by applying the constraint-solver-based strategies, where "successful solving" refers to that a constraint-solver finds a satisfiable solution for these constraints. We observe that for all constraint-solver-based fuzzers, the success rate significantly decreases as the depth approaches 20, e.g., QSYM has 23.6% success rate when the depth reaches 10 while only 15.0% when reaching 20.

Finding 5: Although constraint-solver-based strategies may potentially explore tight and complex constraints, it still becomes less effective when solving deeper constraints.

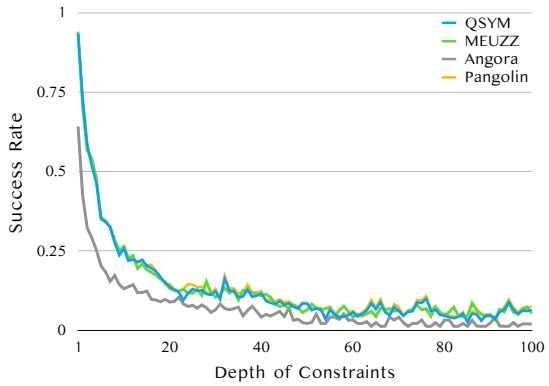

Fig. 6: The success rates of solving constraints at different depths

Dictionary Strategy. We then investigate the potential obstacles which may hinder the future development of the dictionary strategy. Interestingly, we observe that although the *Fuzzer*$_{Dict}$s can slightly outperform the fuzzers with other assisting exploration strategies in terms of the average edge coverage, they fail to achieve consistent performance advantages in each benchmark project. For instance, QSYM and Angora outperform the best performer of *Fuzzer*$_{Dict}$s, i.e., AFL++$_{Dict}$, in nine benchmark projects by 2.1% to 59.2%. We thus infer that the power of the dictionary strategy has not been fully exploited, i.e., randomly selecting tokens to form a dictionary may be essentially deficient. To validate this hypothesis, we set out to evaluate what performance impact can be caused by strategically selecting the tokens to form a dictionary. Specifically, we first randomly selected 10% (268) of *equality constraint*s out of our benchmark suite which can be explored by the fuzzers with all the assisting exploration strategies other than the dictionary strategy. Next, for an edge in CFG corresponding to the failed exploration on an *equality constraint* (e.g., the `False` edge from `memcmp(input, "8BIM")` in Figure 7) when running a seed (e.g., s_1 in Figure 7), we identify its "sibling" edges (i.e., edges under one shared prefix edge defined in [34], such as the `True` edge from `memcmp(input, "8BIM")` in Figure 7). Accordingly, we only collect the associated constant tokens (e.g., `"8BIM"`) and add them into the dictionary of the seed such that it can be randomly inserted to its offsets. At last, we apply AFL++$_{Dict}$ to run the target program. As a result, 93.7% (251) of such *equality constraint*s can be explored by AFL++$_{Dict}$ in three hours, indicating that strategically selecting tokens to form a dictionary can advance the exploration of the program states guarded by *equality constraint*s.

Finding 6: The exploration of program states guarded by the equality constraints can be enhanced by strategically choosing tokens to form a customized dictionary.

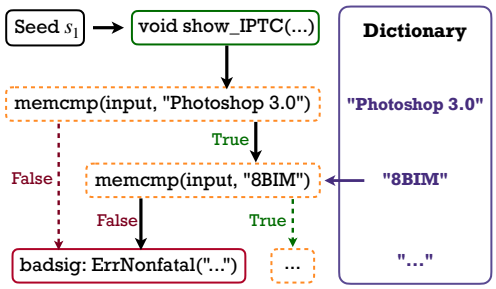

Fig. 7: Strategically building a dictionary for exploring *jhead iptc.c*

E. Discussion

As assisting exploration strategies have been shown to be powerful and yet limited in our study, we then discuss how we could potentially enhance assisting exploration strategies for advancing program state exploration in practice, which essentially demands being lightweight. While it has been widely recognized that improving the taint analysis and constraint solvers can be typically heavyweight to cause potentially excessive efforts [15], [41], [42], strategically selecting tokens to form a dictionary for the dictionary strategy is likely to be lightweight and more practical. Note that while ideally, accurately tracking the offsets of seeds to insert the correct tokens can further improve the effectiveness of the dictionary strategy, it is nevertheless heavyweight as presented in prior work [15], [41], [42]. We thus do not consider accurately tracking the seed offsets to improve over the dictionary strategy.

IV. Customized Dictionary Fuzzing

Motivated by our previous findings, we propose CDFUZZ (***C**ustomized **D**ictionary **Fuzz**ing*) which builds upon the baseline fuzzer AFL a customized dictionary for each seed by accurately selecting tokens.

A. Approach

Figure 8 presents the workflow of CDFUZZ. For each seed in the seed corpus, CDFUZZ first derives its execution path, and then extracts all its constant tokens in *equality constraint*s to form a dictionary for the seed. Next, CDFUZZ randomly selects a token from the dictionary and inserts it into a random

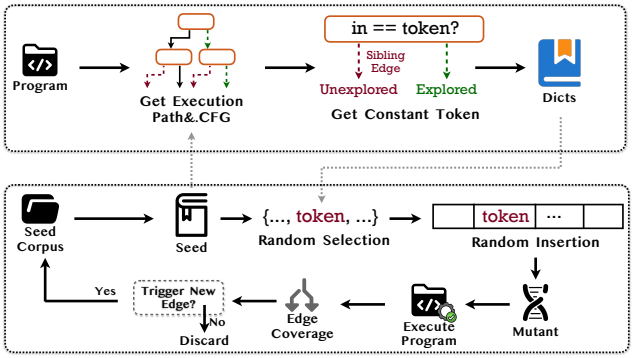

Fig. 8: The workflow of CDFuzz

Algorithm 1 Customized Dictionary Fuzzing

Input: *initialSeed*, *budget*

Output: *seedCorpus*

```
1:  function CustomizedDictionaryFuzzing
2:      CFG ← getCFGFromTargetProgram()
3:      tokens ← getConstantTokensFromCFG(CFG)
4:      path ← getPathBySeed(initialSeed, CFG)
5:      dict ← getValidToken(tokens, path, CFG)
6:      dicts ← ⟦ initialSeed ⇒ {dict}⟧
7:      seedCorpus ← {initialSeed}
8:      while fuzzing time not exceed budget do
9:          for each seed in seedCorpus do
10:             sDict ← dicts⟦seed⟧
11:             token ← randomSelection(sDict)
12:             mutant ← randomInsertion(seed, token)
13:             if mutant has new edges then
14:                 seedCorpus ← seedCorpus ∪ {mutant}
15:                 muPath ← getPathBySeed(mutant, CFG)
16:                 muDict ← getValidToken(tokens, muPath, CFG)
17:                 dicts⟦mutant⟧ ← muDict
18:     return seedCorpus
```

offset to generate a mutant. If running such a mutant upon the target program increases edge coverage, it will be added to the seed corpus.

Instead of maintaining an overall dictionary as in prior dictionary-based approaches [2], [3], [4], [5] , CDFuzz generates a customized dictionary for each seed input. Our intuition is that having a separate dictionary for each seed will (1) allow easier tracking and effective selection of relevant tokens for each seed, and (2) avoid polluting or overloading the dictionary with tokens from other seeds. Algorithm 1 presents the workflow of CDFuzz, which takes an `initialSeed` as input, and performs fuzzing with a given time `budget`. First, we parse the control-flow graph of the target program `CFG` and its corresponding constant tokens following previous work [23] (lines 2–3). Next, we initialize the seed corpus `seedCorpus` by parsing all the constant tokens extracted from the *equality constraints* of the executed path `path` of `initialSeed` via the `getValidToken` function. Specifically, this function extracts constant tokens from the "sibling" edges out of all the edges corresponding to failed exploration on *equality constraints* of `path`. In this way, we form a customized dictionary for `initialSeed` with the collected tokens. For each `seed`, CDFuzz generates a mutant for exploring new program states. To avoid inserting irrelevant tokens into the `seedCorpus`, we only select tokens from the customized dictionary of such `seed` to generate a `mutant` (lines 10–12). Meanwhile, if running such a `mutant` successfully explores new program states, we add it to `seedCorpus` for further exploration (lines 13–14). Similarly, we generate a customized dictionary for this `mutant` by parsing the constant tokens according to its executed path (lines 15–17). For example, assuming that running a seed presented in Figure 7 fails to satisfy the *equality constraint* `memcmp(input, "8BIM")`, CDFuzz then identifies the associated constraint-solving constant token `"8BIM"` while filtering out other irrelevant tokens (e.g., `"Photoshop 3.0"`) to customize a dictionary for further mutations.

TABLE III: The edge coverage results of CDFuzz

Benchmark	AFL$_{Dict}$	AFL++$_{Dict}$	QSYM	REDQUEEN	CDFUZZ
readelf	11,561	11,524	11,386	10,053	**13,276**
nm	5,199	5,270	6,532	5,526	**6,719**
objdump	5,649	5,744	5,999	5,587	**6,539**
size	3,800	4,015	**5,211**	5,002	5,048
strip	6,369	6,598	6,968	5,513	**8,327**
djpeg	2,801	2,628	2,092	2,463	**2,874**
tcpdump	12,001	12,554	10,053	11,471	**14,744**
xmllint	6,862	6,824	6,317	6,771	**7,830**
jhead	801	772	731	612	**865**
pngfix	2,237	1,983	1,976	2,189	**2,342**
tiffinfo	3,831	4,209	3,753	3,981	**4,521**
xmlwf	**4,990**	4,953	4,797	4,879	4,980
tiff2bw	3,503	3,478	2,871	3,615	**4,728**
mutool	2,252	2,295	2,167	2,314	**2,333**
libjpeg-turbo	4,596	4,997	4,722	4,554	**5,001**
libpng	2,092	2,263	2,080	2,014	**2,264**
libxml2	9,621	11,267	10,355	9,001	**12,398**
re2	6,465	6,466	6,379	6,362	**6,470**
jsoncpp	1,455	1,454	1,406	1,444	**1,459**
sqlite3	7,989	6,286	5,013	5,588	**8,209**
bloaty	2,909	3,514	5,595	5,279	**5,771**
Average	5,094	5,195	5,067	4,963	**6,033**
p-value	0.005	0.005	0.005	0.005	-

B. Evaluation

To evaluate CDFuzz, we include the best-performing dictionary-based fuzzer AFL++$_{Dict}$ and constraint-solver-based fuzzer QSYM as well as the input-to-state correspondence fuzzer REDQUEEN for performance comparison. We also include AFL$_{Dict}$ to assess the effectiveness of the customized dictionary by CDFuzz since they only differ in the adopted dictionaries. Similar to the setup in Section III-B, we run each experiment five times to obtain the average result within 24 hours. Note that the dictionary for each seed is built on-the-fly so the time cost is included in the overall running time (24 hours). Prior to running Algorithm 1, CDFuzz first builds CFG and collects constant tokens (this process only incurs roughly 10 seconds overhead per benchmark which is minimal compared to `afl-clang-fast`). We further present the details of compilation cost in our *GitHub* page [20] due to the page limit.

1) Result and analysis: Table III presents the edge coverage results of the studied approaches on top of all the benchmark projects. In general, by only differing the adopted dictionaries, CDFuzz significantly outperforms AFL$_{Dict}$ by 18.4%

(6,033 vs. 5,094 explored edges), indicating the effectiveness of our proposed customized dictionary. Moreover, CDFuzz outperforms AFL++$_{Dict}$ by 16.1% (6,033 vs. 5,195 explored edges), QSYM by 19.1% (6,033 vs. 5,067 explored edges), and REDQUEEN by 21.6% (6,033 vs. 4,963 explored edges) averagely. The results suggest that CDFuzz can significantly improve the effectiveness of the dictionary strategy. We also perform the Mann-Whitney U test [37] to illustrate the significance of CDFuzz. The fact that the p-value of CDFuzz comparing with AFL$_{Dict}$ in terms of the average edge coverage is 0.00503 indicates that CDFuzz outperforms AFL$_{Dict}$ significantly ($p < 0.05$).

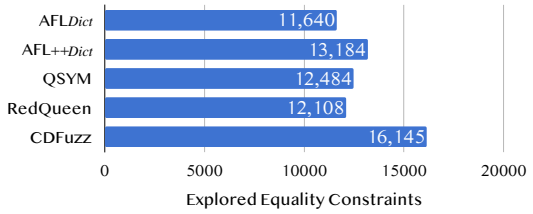

Fig. 9: The explored *equality constraints* of each studied fuzzer

We further investigate how the fuzzers perform on exploring *equality constraints*, as in Figure 9. Specifically, CDFuzz outperforms the runner-up performer QSYM by 29.3% in terms of exploring *equality constraints* (16,145 vs. 12,484). The results indicate that CDFuzz can significantly improve the power of exploring *equality constraints* for grey-box fuzzing.

Figure 10 presents the edge coverage trends of the studied approaches in each benchmark within 24 hours. Overall, CDFuzz outperforms all studied fuzzers significantly in most of the benchmarks (except *xmlwf*, and *size*). Specifically, CDFuzz outperforms the best-performing AFL++$_{Dict}$ of our study in all benchmark projects in terms of edge coverage, e.g., CDFuzz outperforms AFL++$_{Dict}$ by 26.2% in *strip*. Moreover, although AFL$_{Dict}$ and QSYM achieve higher edge coverage than CDFuzz in *xmlwf*, and *size* respectively, their performance gaps are rather limited, i.e., CDFuzz underperforms AFL$_{Dict}$ in *xmlwf* by 0.2%, and QSYM in *size* by 3.1%. Such results altogether indicate that CDFuzz can achieve quite robust edge coverage performance.

2) Bug finding capability for bugs in the wild: To evaluate the bug-finding capability, we apply CDFuzz on our original benchmark suite and randomly select 10 additional real-world open-source projects (stars $>$ 100) from *GitHub* following prior evaluations [28], [15], [43], [3]. We also include all the grey-box fuzzers with dictionary strategy and QSYM, MEUZZ, PANGOLIN, Angora, REDQUEEN in the evaluation of bug finding capacity. To identify unique bugs, we first compile the selected projects with two additional sanitizers [44], [45] to trigger crashes as possible. Next, we derive the unique crashes based on whether they incur unique execution paths following existing work [2], [3], [46], [4], [16], [31], [5]. Finally, we manually analyze each unique crash to derive unique bugs. All bugs are categorized based on their root causes (their details are available in our *GitHub* page [20]). Table IV presents the unique bugs exposed by our approach. CDFuzz has exposed 37 previously unknown bugs where 30 of them cannot be exposed by other studied fuzzers within the given time limit. Moreover, nine of them have been confirmed and seven have been fixed by the corresponding developers. The results suggest that CDFuzz is more effective than other fuzzers in terms of exposing real-world bugs. In particular, we list two examples of the exposed bugs below to illustrate the importance of the bugs found by CDFuzz. We also demonstrate the details of all exposed bugs in our *GitHub* page [20] with their report links due to page limit.

TABLE IV: The unique bugs explored by CDFuzz

Project	Bug Type	Number	Status
bison	Use-of-uninitialized-value	1	*reported*
objdump	Infinite loop	1	*confirmed and fixed*
bsdtar	Use-of-uninitialized-value	1	*reported*
jasper	Assertion failure	1	*confirmed and fixed*
lou_translation	Infinite loop	1	*confirmed and fixed*
libtiff	Use-of-uninitialized-value	7	*reported*
objcopy	Use-of-uninitialized-value	1	*confirmed and fixed*
jhead	Use-of-uninitialized-value	6	*reported*
precomp	Bad-malloc	1	*reported*
nm	Memory leaked	1	*confirmed*
zziplib	Stack buffer overflow	1	*reported*
	Stack buffer overflow	1	*reported*
jpeginfo	Heap-buffer-overflow	1	*confirmed and fixed*
	Use-of-uninitialized-value	1	*confirmed*
cmix	Alloc-dealloc-mismatch	1	*confirmed and fixed*
	Memcpy-param-overlap	1	*confirmed and fixed*
	Use-of-uninitialized-value	1	*reported*
bento4	Allocation-size-too-big	1	*reported*
	Out-of-memory	2	*reported*
	Memory leaked	1	*reported*
	Heap-buffer-overflow	3	*reported*
	Segmentation fault	1	*reported*
	Heap-use-after-free	1	*reported*

Heap-buffer-overflow bug in project *jpeginfo*. Figure 11 shows the code snippet for a heap-buffer-overflow bug in project *jpeginfo*, where in line 9, six bytes from `EXIF_IDENT_STRING` are copied to `cmarker->data` without any length checking, leading to a buffer overflow. To expose this bug, the *equality constraint* `cmarker->marker == EXIF_JPEG_MARKER` should be satisfied. In our evaluation, only CDFuzz reaches the branch guarded by such *equality constraint*. The developer fixed the bug [47] by adding buffer length checking statements from line 7 to line 8. They also replied to our bug report as follows:

> *"Thanks, looks like memcmp() may have read past end of the buffer in some circumstances."*

Memcpy-param-overlap bug in project *cmix*. We have also reported a memory-overlapping bug in *cmix* only exposed by CDFuzz. The corresponding developers have fixed this bug after receiving our report [48]. Figure 12 shows the corresponding buggy code snippet where in line 7, the memory content is copied from source `W->Letters[i+2]` to sink `W->Letters[i+1]` via `memcpy`. As the source and sink addresses differ by a single byte, it causes potential

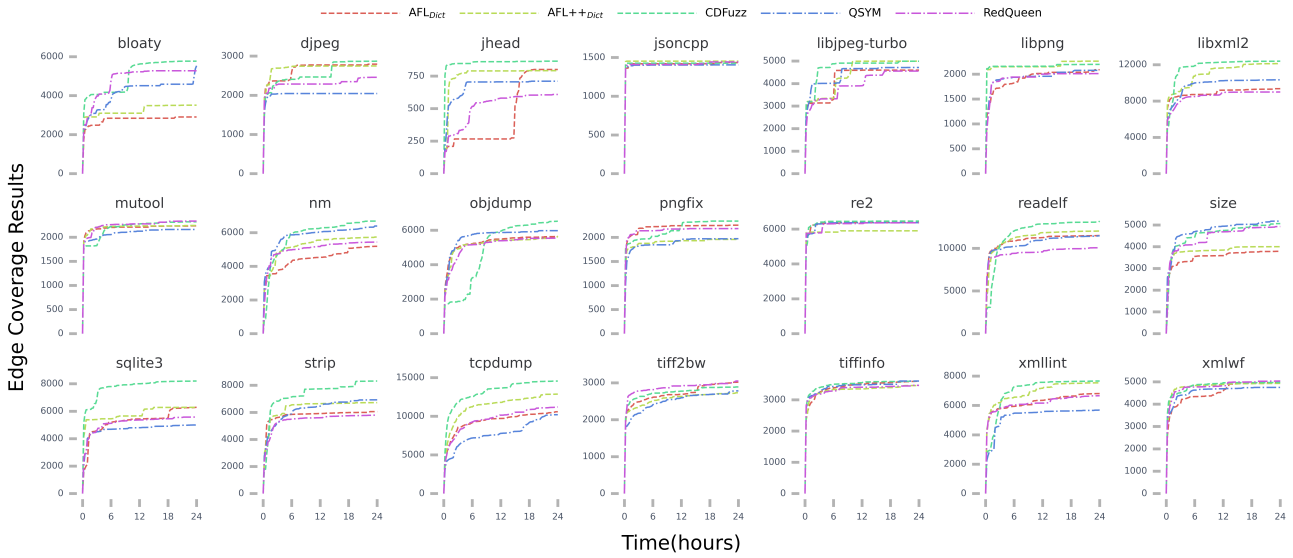

Fig. 10: Edge coverage of CDFuzz over time

```
#define EXIF_JPEG_MARKER JPEG_APP0+1
#define EXIF_IDENT_STRING "Exif\000\000"
 [+] #define EXIF_IDENT_STRING_LEN 6
...
while (cmarker) {
     [-] if (cmarker->marker == EXIF_JPEG_MARKER)
     [+] if (cmarker->marker == EXIF_JPEG_MARKER &&
     cmarker->data_length >= EXIF_IDENT_STRING_LEN)
     if (!memcmp(cmarker->data,EXIF_IDENT_STRING,6))
         exif_marker = cmarker;
     cmarker = cmarker->next;
}
```

Fig. 11: A heap-buffer-overflow bug in *jpeginfo*

```
1  void ConvertUTF8(Word *W) {
2      for (int I = W->Start; i < W->End; i++) {
3          U8 c = W->Letters[i+1] + ((W->Letters[i+1]<0xA0)?0
       x60:0x40);
4          if (W->Letters[I] == 0xC3 && (IsVowel(c) || (W->
       Letters[i+1]&0xDF) == 0x87)) {
5              W->Letters[i] = c;
6              if (i+1 < W->End)
7
           [-] memcpy(&W->Letters[i+1], &W->Letters[i+2],
8                      W->End-i-1) ;
9
           [+] memmove(&W->Letters[i+1], &W->Letters[i+2],
10                     W->End-i-1) ;
11             W->End--;
12       }}}
```

Fig. 12: A memcpy-param-overlap bug in *cmix*

overlapped memory, i.e., an undefined behavior in C/C++ programming. The function `memcpy` does not guarantee proper handling of overlapping memory regions. In contrast, `memmove` ensures accurate data replication in such cases. In our evaluation, only CDFuzz generates seeds that expose this defect by satisfying the *equality constraint* in line 4, i.e., `W->Letters[i]==0xC3` and `IsVowel(c)`. The developers also commented on our report:

> *"Thanks for the bug report, and the suggested fix! Changing to memmove fixed this."*

V. Threats to Validity

Internal validity. One threat to internal validity lies in the implementation of the studied techniques in our evaluation. To reduce this threat, we reused all the source code from the original projects directly in our implementation with best effort. When implementing the dictionary strategy, we proposed an automatic approach to extract tokens in a LLVM pass following prior work [23] to reduce potential bias caused by user-provided tokens. Moreover, all the student authors manually reviewed the code of all studied fuzzers including CDFuzz to ensure their correctness and consistency.

External validity. The threat to external validity lies in the subjects and benchmarks. To reduce this threat, we have selected 9 representative state-of-the-art fuzzers which cover mainstream types of assisting exploration strategies, including dictionary-based fuzzers, input-to-state-correspondence-based fuzzers, gradient-based fuzzers, and SMT-solver-based fuzzers. We also collect 21 frequently used projects from their original papers as our benchmark suite.

Construct validity. The threat to construct validity mainly lies in the metrics used. To reduce this threat, we adopted the most popular metrics in fuzzing, i.e., edge coverage following [4], [2], [3], [5], [49], to reflect the performance of different studied techniques. Furthermore, we evaluated the effectiveness of our approach in terms of the number of unique crashes.

VI. Related Work

A. Fuzzing

Most existing fuzzers [5], [50], [46], [28], [25], [51], [52], [53] use code coverage information to improve the efficiency

of fuzzing. In particular, AFL [2] provides the fundamental framework for coverage-guided fuzzers. Accordingly, Fioraldi et al. [3] integrated several techniques (e.g., taint tracking) to enhance the ability of exploring program states for AFL. She et al. [34] proposed NEUZZ, which leverages the power of neural network models to explore unknown edges. Pham et al. [54] proposed SGF which generates seeds on the virtual structure of the file rather than on the bit level to improve fuzzing efficacy. Meanwhile, *AFLFast* uses Markov Chain to schedule seeds for exploring program states [46]. FAIRFUZZ focused on rare branches for its exploration [5]. Zeror is a coverage-sensitive tracing and scheduling fuzzing framework that uses zero-overhead instrumentation and a schedule strategy between different instrumentation for AFL-based fuzzers [55]. Recently, researchers also pay attention to exploring program states by focusing on the diversity of the program behaviors. Nguyen et al. [56] introduced BeDivFuzz to schedule the mutation strategy towards the validity and diversity of program behaviors based on the received program feedback. Liang et al. [28] proposed PATA to mutate the influencing input bytes by leveraging the power of diverse explored program paths. Yan et al. [57] introduced a new approach PathAFL to reduce the tracing granularity of an execution path for exploring program states. QATest adopts a new coverage guidance and seed schedule strategy [58] for question-answering systems. EMS utilizes historical explorations to identify mutators that can trigger unique paths and crashes [59]. By adopting an automatic dictionary generation strategy, FuzzingDriver generates dictionary tokens for coverage-based grey-box fuzzers via parsing the original code [23]. Compared to FuzzingDriver, CDFUZZ schedules existing tokens in dictionaries for different seeds in runtime instead of generating tokens before fuzzing.

To explore program states guarded by complicated constraints, hybrid fuzzing techniques are proposed to combine constraint solvers with grey-box fuzzers. Majumdar et al. [7] presented hybrid fuzzing to interleave random fuzzing with constraint solver for deep exploration of program state space. Driller leverages fuzzing and selective concolic execution with constraint solver in a complementary manner to explore program states [60]. Chen et al. [16] leveraged the power of gradient descent to solve the constraints in the target program. QSYM optimistically solves constraints and prunes uninteresting basic blocks during fuzzing [18]. Chen et al. [26] introduced a machine-learning-based seed scheduling strategy for hybrid fuzzing to explore program states efficiently. Huang et al. [22] utilized polyhedral path abstraction to facilitate constraint solving. CONFETTI fuzzes Java programs by combining fuzzing with taint tracking and concolic execution with constraint solver [61]. Meanwhile, *chopped symbolic execution* [62] leverages various on-demand static analyses at runtime to automatically exclude code fragments while resolving their side effects to improve the efficiency of constraint solving. Compared to constraint solver and dynamic taint tracking adopted by hybrid fuzzing, CDFUZZ generates a customized dictionary for each seed via a lightweight static analysis. Our experiments show that CDFUZZ outperforms the state-of-the-art constraint-solving fuzzers.

B. Studies on Fuzzing

Many empirical studies [29], [31], [13], [63], [25], [64], [65], [66] on fuzzing reveal various insights for improving fuzzing techniques. Donaldson et al. [67] investigated a variety of fuzzing techniques, including coverage-guided fuzzing with and without custom mutators to test compilers and processing tools for the graphics shading languages. Wu et al. [13] conducted a study on *Havoc* fuzzing strategy and demonstrated that it largely outperforms other strategies. Böhme et al. [68] performed a study on discussing the reliability metrics for evaluating the effectiveness of different coverage-based fuzzers. They also study the scalability issues of fuzzing in vulnerability discovery [69]. FuzzBench is a open-source platform proposed for evaluating fuzzers to facilitate reliable and reproducible evaluation results [29]. Herrera et al. [70] systematically investigated and evaluated how seed selection affects a fuzzer's ability to expose vulnerabilities in real-world systems. Klees et al. [31] provided multiple guidelines about how to evaluate the effectiveness of different fuzzers. In this paper, we conduct the first comprehensive study to investigate how assisting exploration strategies perform in exploring program states and reveal various findings to facilitate future research.

VII. CONCLUSION

In this paper, we investigated the strengths and limitations of assisting exploration strategies for exploring program states. We first conduct an extensive evaluation to investigate how assisting exploration strategies perform in exploring program states. The evaluation results suggest that dictionary strategy can be close to or even slightly more effective than other techniques. Next, we investigate their limitations and find that the dictionary strategy is most promising to be improved. Inspired by our findings, we present a lightweight approach namely CDFUZZ which customizes the dictionary for each seed. Our evaluation results show that CDFUZZ outperforms the best performer in our study by 16.1% in terms of edge coverage. CDFUZZ also exposes 37 previously unknown bugs where nine of them have been confirmed and seven of them have been fixed by the corresponding developers.

DATA AVAILABILITY

The data and code are available at *GitHub* [20] for public evaluation.

VIII. ACKNOWLEDGEMENT

This work is partially supported by the National Natural Science Foundation of China (Grant No. 62372220) and Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant. It is also partially supported by the Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang (Grant No. TD2019001) and Ant Group Research Fund.

REFERENCES

[1] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] Q. Xiao, Y. Chen, C. Wu, K. Li, J. Mao, S. Guo, and Y. Shi, "pbse: Phase-based symbolic execution," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 133–144.

[9] 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.

[10] "Custom mutators in afl++," https://aflplus.plus/docs/custom_mutators/, 2023.

[11] "Custom mutators in libfuzzer," https://github.com/google/libprotobuf-mutator, 2023.

[12] 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.

[13] M. Wu, L. Jiang, J. Xiang, Y. Huang, H. Cui, L. Zhang, and Y. Zhang, "One fuzzing strategy to rule them all," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022.

[14] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 724–735.

[15] 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.

[16] 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.

[17] B. Mathis, R. Gopinath, and A. Zeller, "Learning input tokens for effective fuzzing," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 27–37.

[18] 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.

[19] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[20] "Github repository. 2023. fuzzing-empirical-study," https://github.com/SophrosyneX/Fuzzing-empirical-study, 2023.

[21] C. Kaner, J. Falk, and H. Q. Nguyen, *Testing computer software*. John Wiley & Sons, 1999.

[22] H. Huang, P. Yao, R. wu, Q. Shi, and C. Zhang, "Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction," 05 2020, pp. 1613–1627.

[23] A. A. Ebrahim, M. Hazhirpasand, O. Nierstrasz, and M. Ghafari, "Fuzzingdriver: the missing dictionary to increase code coverage in fuzzers," *arXiv preprint arXiv:2201.04853*, 2022.

[24] 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.

[25] 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*, 2022, pp. 847–858.

[26] 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.

[27] "Aflplusplus redqueen mode," https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.cmplog.md, 2023.

[28] 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 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 1–17.

[29] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "Fuzzbench: an open fuzzer benchmarking platform and service," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 1393–1403. [Online]. Available: https://doi.org/10.1145/3468264.3473932

[30] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[31] 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.

[32] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, "Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization." in *NDSS*, 2020.

[33] 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.

[34] 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.

[35] B. Shastry, M. Leutner, T. Fiebig, K. Thimmaraju, F. Yamaguchi, K. Rieck, S. Schmid, J.-P. Seifert, and A. Feldmann, "Static program analysis as a fuzzing aid," in *Research in Attacks, Intrusions, and Defenses: 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18–20, 2017, Proceedings*. Springer, 2017, pp. 26–47.

[36] "Codeql github page," https://codeql.github.com, 2022.

[37] T. W. MacFarland and J. M. Yates, "Mann–whitney u test," in *Introduction to nonparametric statistics for the biological sciences using R*. Springer, 2016, pp. 103–132.

[38] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[39] K. R. Irvine and L. B. Das, *Assembly language for x86 processors*. Prentice Hall, 2011.

[40] G. Lee, W. Shim, and B. Lee, "Constraint-guided directed greybox fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 3559–3576. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/lee-gwangmu

[41] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, "Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery," in *2019 IEEE symposium on security and privacy (SP)*. IEEE, 2019, pp. 769–786.

[42] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "Greyone: Data flow sensitive fuzzing." in *USENIX Security Symposium*, 2020, pp. 2577–2594.

[43] 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.

[44] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 309–318.

[45] E. Stepanov and K. Serebryany, "Memorysanitizer: fast detector of uninitialized memory use in c++," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2015, pp. 46–55.

[46] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*,

ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1032–1043. [Online]. Available: https://doi.org/10.1145/2976749.2978428

[47] "Github repository. 2023. jpeginfo heap-buffer-overflow," https://github.com/tjko/jpeginfo/issues/13, 2023.

[48] "Github repository. 2023. cmix memcpy-param-overlap," https://github.com/byronknoll/cmix/issues/54, 2023.

[49] V. J. Manès, S. Kim, and S. K. Cha, "Ankou: Guiding grey-box fuzzing towards combinatorial difference," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1024–1036.

[50] "Libfuzzer – a library for coverage-guided fuzz testing," https://llvm.org/docs/LibFuzzer.html, 2023.

[51] M. Wu, Y. Ouyang, M. Lu, J. Chen, Y. Zhao, H. Cui, G. Yang, and Y. Zhang, "Sjfuzz: Seed and mutator scheduling for jvm fuzzing," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1062–1074.

[52] M. Wu, M. Lu, H. Cui, J. Chen, Y. Zhang, and L. Zhang, "Jitfuzz: Coverage-guided fuzzing for jvm just-in-time compilers," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 56–68.

[53] M. Wu, K. Chen, Q. Luo, J. Xiang, J. Qi, J. Chen, H. Cui, and Y. Zhang, "Enhancing coverage-guided fuzzing via phantom program," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1037–1049.

[54] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1980–1997, 2019.

[55] C. Zhou, M. Wang, J. Liang, Z. Liu, and Y. Jiang, "Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 858–870.

[56] H. L. Nguyen and L. Grunske, "BEDIVFUZZ: integrating behavioral diversity into generator-based fuzzing," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 249–261. [Online]. Available: https://doi.org/10.1145/3510003.3510182

[57] S. Yan, C. Wu, H. Li, W. Shao, and C. Jia, "Pathafl: Path-coverage assisted fuzzing," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 598–609. [Online]. Available: https://doi.org/10.1145/3320269.3384736

[58] Z. Liu, Y. Feng, Y. Yin, J. Sun, Z. Chen, and B. Xu, "Qatest: A uniform fuzzing framework for question answering systems," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[59] C. Lyu, S. Ji, X. Zhang, H. Liang, B. Zhao, K. Lu, and R. Beyah, "Ems: History-driven mutation for coverage-based fuzzing," in *29th Annual Network and Distributed System Security Symposium. https://dx. doi. org/10.14722/ndss*, 2022.

[60] 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.

[61] J. Kukucka, L. Pina, P. Ammann, and J. Bell, "Confetti: Amplifying concolic guidance for fuzzers," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 438–450.

[62] 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.

[63] L. Jiang, H. Yuan, M. Wu, L. Zhang, and Y. Zhang, "Evaluating and improving hybrid fuzzing," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 410–422.

[64] T. Gao, J. Chen, Y. Zhao, Y. Zhang, and L. Zhang, "Vectorizing program ingredients for better jvm testing," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 526–537.

[65] 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*, 2022, pp. 1133–1144.

[66] 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*, 2020, pp. 937–948.

[67] A. F. Donaldson, B. Clayton, R. Harrison, H. Mohsin, D. Neto, V. Teliman, and H. Watson, "Industrial deployment of compiler fuzzing techniques for two gpu shading languages," in *IEEE International Conference on Software Testing, Verification and Validation (ICST'23)*, 4 2023.

[68] M. Böhme, L. Szekeres, and J. Metzman, "On the reliability of coverage-based fuzzer benchmarking," in *44th IEEE/ACM International Conference on Software Engineering, ser. ICSE*, vol. 22, 2022.

[69] M. Böhme and B. Falk, "Fuzzing: On the exponential cost of vulnerability discovery," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 713–724. [Online]. Available: https://doi.org/10.1145/3368089.3409729

[70] 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*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 230–243. [Online]. Available: https://doi.org/10.1145/3460319.3464795