PASTIS: A Collaborative Approach to Combine Heterogeneous Software Testing Techniques

1st Robin David
Quarkslab
Paris, France
rdavid@quarkslab.com

2nd Richard Abou Chaaya
Quarkslab
Paris, France
rabouchaaya@quarkslab.com

3rd Christian Heitman
Quarkslab
Buenos Aires, Argentina
cheitman@quarkslab.com

***Abstract*—The fuzzing research field experienced outstanding advances over the past decade, making it a very effective approach for software testing. Dynamic Symbolic Execution (DSE) also called *whitebox fuzzing* is another approach that also evolved significantly and has the advantage of being able to solve very complex path conditions. Given fuzzing advances, knowing whether DSE is still relevant to help improving fuzzing coverage is an open research question. This paper aims at answering this question by empirically studying software testing techniques in a collaborative environment, and especially combining fuzzing and DSE. We show that in some cases DSE helps uncover some code locations not reached by fuzzers or can speed up the discovery of such coverage. We propose a combination of fuzzing and DSE into an *ensemble fuzzing* framework called PASTIS that helps in circumventing engines inner-working discrepancies. In this line, we propose a few DSE optimizations to maximize its ability to provide relevant inputs to fuzzers. One of them is a coverage strategy called *prefixed-edge*, which is an optimization over edge coverage. Finally, we show the advantages and roadblocks one can meet in trying to combine heterogeneous fuzzers and give insights on how to do it more efficiently.**

***Index Terms*—Software Testing, Greybox Fuzzing, Dynamic Symbolic Execution, Ensemble Fuzzing**

I. Introduction

Context: Software testing is now crucial to uncover bugs and vulnerabilities. To that end, multiple automated testing techniques like fuzzing are used. This approach has been extensively studied in the literature [1], [2] and improved over the last few years. Fuzzing relies on executing as many iterations as possible of the PUT (Program Under Test) over different inputs generated with pseudo-random mutations and possibly with a the help of structure model or grammar. Both execution and input generation algoritms have been improved over time to explore deeper program states.

Dynamic Symbolic Execution (DSE) is a formal approach also used for program exploration and testing. Advances performed in this research area made it a functional approach used in state-of-the-art software testing tools. DSE principle is to model precisely each instruction's side-effects to be able to track input propagation in the program and express branching conditions as first-order logic formulas.

Problem: While fuzzing is empirically effective, it tends to cover shallower states. In comparison, DSE is slower, but is theoretically able to cover deeper states by solving complex branching conditions or complex code constructs [3]. Given the recent fuzzing improvements and the slow nature of DSE, one may wonder if using DSE can improve PUT coverage compared to fuzzing. This raises the first research question **RQ1**: Can DSE help a fuzzing engine in a collaborative environment? By extension, their approach being extensively very different, it is unclear whether fuzzing and DSE can work together efficiently and whether this combination may outperform both approaches used independently. Therefore the second research question is **RQ2**: Can a collaborative approach like ensemble fuzzing reach better coverage than the sum of its parts?

Intuition: Our approach is based on two intuitions commonly shared in the software testing research field. First, no fuzzer or input generation algorithm is universally better on any kind of target due to the wide range of software and the contrasting input format that can be encountered. Second, in a DSE, the SMT solving phase is very costly and should be avoided whenever possible. This implies that the DSE should not attempt to solve a branching condition if it has already been covered by another test engine.

Goal and Challenges: The goal is to combine greybox fuzzing and DSE to leverage their respective strengths and reach better coverage than either of these approaches on its own, or at least, obtaining the same coverage faster. Challenges are threefold. First, one needs to deal with the implementation discrepancies of various engines, such as input formats and execution speed. Second, input generation throughput is a challenge as input flooding might alter normal behavior of engines. The last challenge is to combine them in an asynchronous manner so that no one is blocking or slowing down the others.

Our Approach: It combines heterogeneous test engines by solely sharing test cases (inputs). Each engine then decide whether to drop it or not. If the input triggers a new program behavior regarding engine's coverage metric the input is kept, otherwise it is discarded. Being significantly slower than fuzzing, DSE should replay each input it receives at a satisfying speed to update its coverage and deciding whether to keep the input. We designed an *ensemble fuzzer* combining greybox fuzzing and whitebox fuzzing (DSE) built around a broker that performs seed sharing and aggregates the resulting corpus and data.

Contributions: Ensemble fuzzer with seed synchronisation have already been studied and implemented in the literature. Nonetheless, few tries to combine greybox fuzzing and DSE which are working differently and at different paces. Furthermore, very few of them study the practical impact and issues encountered when combining them. This main contributions of our work are as follow:

- we propose `Pastis` an open-source *ensemble fuzzer* framework combining disparate testing approaches in a fully asynchronous manner. Designed to be flexible, any software testing engine can be integrated as long as it produces tests-cases and is able to absorb arbitrary ones dynamically[1];
- we introduce TritonDSE a whitebox fuzzing (DSE) framework based both on Triton, for symbolic execution, and QBDI, a Dynamic Binary Instrumentation tool, for test-case replay. Through experiments, we tuned its configuration parameters (solving strategy, coverage metric, whether to explore symbolic pointers, etc.) for improved performance in the context of ensemble fuzzing;
- we propose a new coverage metric called *prefixed-edge* which is as precise as edge, but more scalable as it saves worthless SMT queries;
- we show that ensemble fuzzing either in half-duplex (seed aggregation) or full-duplex (seed sharing across engines) can outperform fuzzers used alone, especially on short-term campaigns;
- we empirically show that DSE can sometimes help greybox fuzzers by unlocking them on specific branches, thus improving coverage or reaching such a coverage faster;
- we conducted an experimental study of the performance impact of ensemble fuzzing showing its strengths but also its weaknesses, as it can drive the coverage discovery down by means of input overproduction. It debunks the intuition that maximal seed sharing necessarily leads to better coverage.

II. Background

A. Fuzzing

Fuzzing, and more especially greybox fuzzing [4], is a testing technique composed of two main components. First, an instrumentation mechanism aims at obtaining feedback about the program's execution. It can be inserted at compile-time if source code is available or at runtime and usually provides coverage as feedback. Hence, these fuzzers are deemed *coverage-guided* as they aim at maximizing coverage. This feedback is also used by the second component; the input generation algorithm to drive the fuzzing process. It can be mutational, which means inputs are generated by applying pseudo-random mutations or it can use some format description like a grammar or a model to create new test cases. In this latter case the generation is called structured. The input generation algorithm uses execution feedback to further mutate a promising input. Most greybox fuzzers are coverage-guided so they keep mutating inputs that generate new coverage. The fuzzing algorithm is an infinite loop that generates new test cases and provides them as input to the instrumented target.

[1]The core of `Pastis` is simply a network message exchange format, it could theoretically be extended to any programming language or instrumentation technique

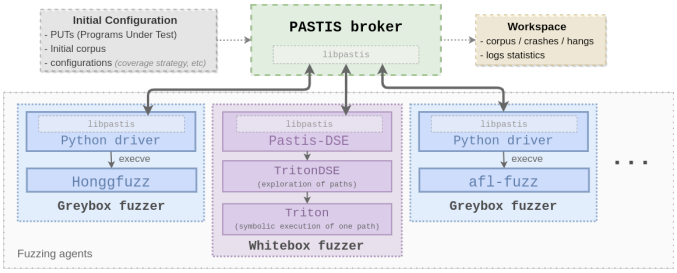


Fig. 1: `Pastis` ensemble fuzzer overview

B. Dynamic Symbolic Execution

Dynamic Symbolic Execution, also called *whitebox fuzzing* [5], is a testing technique working on both a concrete state and a symbolic state. The former represents the concrete values of registers and memory cells during execution. The latter is their symbolic representation whose values are computed by means of evaluating instructions through their semantic modeling, usually performed using an Intermediate Representation (IR). In this context, symbolic values are the bytes of the current input in memory.

By evaluating a program over symbolic input, the DSE follows a path and constructs a path predicate which is the conjunction of logical expressions representing branching conditions expressed as constraints over the symbolic variables on this path. In order to cover new branches, the DSE negates these constraints by encoding them as first-order logic formula, usually on bitvectors. Each formula (query) is provided to an SMT solver, which, if satisfiable, provides a model representing a new input satisfying all the conditions. By negating and solving every uncovered branching condition a DSE is able to cover the program.

Some symbolic execution engines like KLEE [6] work on the LLVM-IR obtained usually from source code, while most engines, like Triton [7], angr [8] or Maat [9] work directly at binary-level.

III. Our Approach

A. Overview

Our ensemble fuzzer is designed around a central component, the broker, holding the PUT variants instrumented for each engine, as well as the initial corpus. When a test engine connects to the broker, it announces its underlying fuzzing engine. Consequently, the broker sends it the appropriate PUT variant and the initial corpus. Figure 1 shows the general architecture. Once running, an engine can submit test-cases to the broker along with its type: input, crash or hang.

The ensemble fuzzer can operate in two main modes:

- $\overrightarrow{\cup}$, half-duplex mode, where the broker aggregates test-cases but does not forward them to any other engine. The resulting coverage is thus the union coverage of individual engines.
- $\overrightarrow{\cap}$, full-duplex mode, where the broker shares submitted test-cases with all the other engines. The final coverage is consequently the result of engines collaborating with one another.

In both modes, engines are treated equally regardless of their underlying inner workings. To that end, each engine has to decide whether to keep or discard an input depending on its own coverage metric [10].

The resulting campaign coverage is computed *a-posteriori* on the corpus generated. While it could rely on `gcov` or `llvm-profile`, coverage is computed using QBDI [11]. All design choices of the ensemble fuzzer have been done with the closed-source scenario in mind.

B. Implementation

The algorithm has been implemented in `Pastis`, an ensemble fuzzing framework. Its articulated around *libpastis*, a Python library providing network communication functionalities between components. It uses a message queuing library called `ZeroMQ`[2]. Another component is the broker, which relies on *libpastis*, acts as a server to manage the fuzzing campaign.

Adding support for a given engine requires implementing a *driver* in Python making use of *libpastis* to communicate with the broker. The driver should first connect to the broker and to launch the underlying fuzzer with the appropriate options received by the broker. Once running, it should transmits inputs received from the broker to the fuzzer and vice-versa.

Our tests focus on Linux targets, however, this framework can target Windows or MacOS applications as long as engines work on these platforms. The current version support AFL++ 4.02a, Honggfuzz 2.5 and TritonDSE 0.1. AFL++ supports a distributed mode to receive new inputs. Nonetheless Honggfuzz does not have this feature, so it has been patched to support dynamic insertion of new inputs. For that, the reading of dynamic input directory has been introduced in the global fuzzing loop so that Honggfuzz checks periodically the directory for new inputs. The whole `Pastis` framework with all its drivers are now open-source[3].

C. TritonDSE

TritonDSE relies on the existing Triton framework [7] which is a low-level DSE engine library which works on a single trace for which the user has to provide every instruction to execute symbolically. However, it does not provide all the expected functionalities of a fully-featured whitebox fuzzer. To that end, TritonDSE has been developed at the top. While it could have been made *concolic* by combining it with a DBI like QBDI, the initial concrete state is instantiated by manually loading the program and its libraries. All operating system interactions are either modeled with symbolic stubs or skipped. As a consequence, TritonDSE can generate unsound results. This framework provides the following features:

- program loading: sole executable or also with its shared libraries using `cle`[4];
- symbolic emulation until reaching a halt instruction or a libc `exit()` call;
- automatic solving of uncovered branching conditions, thanks to the coverage strategy among block $\mathcal{B}$, edge $\mathcal{E}$, prefixed-edge $\mathcal{PE}$ or path $\mathcal{P}$;
- automatic generation of new inputs with solved branching conditions;
- enumeration of symbolic reads, writes or symbolic dynamic jumps;
- memory segmentation with permissions with a basic allocator.

This component is now also available as a standalone open-source project [5].

D. Prefixed-Edge Strategy: $\mathcal{PE}$

This coverage strategy is based on the observation that: it is irrelevant to try solving twice an edge twice if the path leading to that edge is the same, as formulas will be identical. However, in pure edge coverage, the algorithm must try to solving whenever an uncovered edge is reached in the execution trace. In $\mathcal{PE}$ strategy, the path taken to reach an edge is also considered to decide to solve or not. Therefore, if an edge is reached via a path previously taken and for which an unsatisfiable solving was already attempted, no duplicate solving will be done. Once the edge is solved, no further attempt will be performed even though the edge might be reached by a new prefix path. This strategy is not meant to obtain a more granular coverage, but to optimize solving attempts. This strategy is evaluated in benchmarks V-A.

IV. Experimental Settings

A. Dataset

We chose at random a set of 8 targets for benchmarking: *harfbuzz*, *freetype*, *libjpeg*, *libpng*, *vorbis*, *openthread* and *zlib* taken from fuzzbench [12], a reference benchmark. The last target is *CycloneTCP*[6] a TCP/IP stack used in embedded systems and IoT.

The harness built for *CycloneTCP* is a simple HTTP server with a basic stack configuration (IPv4, ARP, ICMP, TCP, DHCP) representative of configurations that can be found in many embedded and IoT devices. The input format is a sequence of ethernet frames separated by a marker. While the stack normally involves multiple threads, the harness breaks down the execution to a single-threaded application. The input is read on `stdin` where each frame is split and injected in the stack sequentially.

[2] https://zeromq.org/
[3] https://github.com/quarkslab/pastis
[4] https://github.com/angr/cle
[5] https://github.com/quarkslab/tritondse
[6] https://www.oryx-embedded.com/products/CycloneTCP

Benchmarks involve 3 engines: AFL++, Honggfuzz and TritonDSE. The targets are compiled for the three engines. TritonDSE use the "vanilla" target without instrumentation.

B. Benchmark Methodology

Benchmarks are split in two phases. Benchmark #1 aims at selecting the best preset for TritonDSE in order to make it as compelling as possible for benchmark #2 which performs ensemble fuzzing tests with greybox fuzzers. Coverage is the reference metric to compare test engines performances [12] and serves as basis to evaluate results.

V. Benchmark #1: Hyperparameter Selection

These benchmarks run TritonDSE sole instances with different configuration in 8 hours campaigns, on a 2.452 Ghz processor and 126Gb of RAM.

A. benchmark #1a: Coverage Strategy

The benchmark aims at selecting the best coverage strategy and more specifically, the most cost-efficient in terms of SMT solving and corpus size. In the benchmark, four TritonDSE instances were launched using the four different coverage strategies implemented: block $\mathcal{B}$, edge $\mathcal{E}$, prefixed-edge $\mathcal{PE}$ and path $\mathcal{P}$. Results are shown in Table I. The coverage column shows the number of edges discovered by the exploration in addition to the ones generated by the initial corpus. The input column shows the number of test-cases generated and by extension the number of SAT formulas.

	Coverage					**#Input (SAT)**			
	base	$\mathcal{B}$	$\mathcal{E}$	$\mathcal{PE}$	$\mathcal{P}$	$\mathcal{B}$	$\mathcal{E}$	$\mathcal{PE}$	$\mathcal{P}$
cyclone	844	+275	+305	+304	+223	105	125	126	26720
freetype	2630	+545	+633	+569	+523	113	136	136	26864
harfbuzz	3017	+222	+335	+312	+429	107	236	189	4209
libjpeg	772	+60	+66	+66	+66	104	128	129	18246
libpng	335	+75	+95	+96	+326	28	42	43	29835
openthread	1095	+0	+0	+0	+0	0	0	0	0
vorbis	883	+117	+136	+137	+95	25	45	46	57669
zlib	66	+19	+19	+19	+21	3	3	3	4
Total	9642	10955	11231	11145	**11325**	485	**715**	672	163547

TABLE I: Benchmark #1a: Coverage Strategy

For most targets, $\mathcal{P}$ strategy has not finished executing every input at the end of the campaign resulting in lower coverage. This strategy generates a very large amount of inputs for scarce edge coverage improvement. This strategy is thus impractical in an ensemble fuzzing context. Then $\mathcal{PE}$ provides very similar results than $\mathcal{E}$. Its coverage is significantly better for *harfbuzz*. While not visible on table, $\mathcal{PE}$ reduces the number of SMT queries by 34% on average compared to $\mathcal{E}$. Given the time overhead this implies, the $\mathcal{PE}$ is chosen as the default coverage metric.

B. benchmark #1b: SMT solving

The SMT solver used in a DSE engine plays an important role as a significant amount of time is spent solving queries. The Triton library supports Z3 [13] and Bitwuzla [14] which both support the `QF_BV` logic. This benchmark aims at evaluating the most suitable solver. No solving timeout is set for this study. Table II shows the results obtained. The "avg query" column gives the average solving time for queries.

	SAT		**UNSAT**		**avg query (s)**	
	Z3	Bitwuzla	Z3	Bitwuzla	Z3	Bitwuzla
cyclone	109	127	5423	4281	0.42	0.21
freetype	126	136	415	559	0.41	0.17
harfbuzz	213	205	7175	8040	3.98	3.54
libjpeg	132	129	3549	1783	8.86	0.28
libpng	46	43	161	180	0.03	0.00
openthread	0	0	17	17	0.02	0.00
vorbis	41	46	1215	1731	0.56	0.18
zlib	4	3	12	7	0.03	0.01

TABLE II: Benchmark #1b: SMT solving

Results presented in Table II show a great target disparity in terms of solving as multiple targets explore the possible paths quickly while others like *harfbuzz*, make both solvers spending 98% of the whole campaign time solving queries (∼7h50m). Bitwuzla outperforms significantly Z3. The average query time is almost twice as fast. The resulting solving time ranges from 2x faster to 54x on *libjpeg*. While both solvers might provide similar results on smtlib2 files, APIs are rather different and Triton spends a significant amount of time converting formulas into Z3 structures, which is taken into account in the solving time.

Given the results, Bitwuzla is selected as the default solver for the remaining experiments.

C. benchmark #1c: Symbolic Exploration

A DSE algorithm explores a program by solving uncovered branching conditions. However, some code constructs, like switch statements, are slightly more complex to cover as they rely on jump address tables. In this scenario the jump address is not symbolic but the index in the table can be. Therefore, a DSE engine has to enumerate the index value if it depends on the input. Performing pointer coverage by enumerating its values can help covering new locations. TritonDSE implements symbolic reads and symbolic writes enumeration. This benchmark evaluates the gain of symbolic pointers enumeration in Table III. Base indicates the baseline results, then, symbolic reads/writes shows the relative improvement against base. The SMT column is the total number of queries, input the number of test-cases thus the number of SAT formulas and *cov* the relative coverage improvement expressed in edges.

	base			**symbolic reads**			**symbolic writes**		
	SMT	input	cov	SMT	input	cov	SMT	input	cov
cyclone	1506	125	305	+259	+237	+0	+55	+48	+0
freetype	667	137	570	+71	+58	+105	-	-	-
harfbuzz	14793	380	507	-4083	+10535	+41	-672	+280	+17
libjpeg	825	107	66	+435	+2199	+3	+289	+283	+0
libpng	211	43	96	+372	+351	+1	-	-	-
openthread	17	0	0	+0	+0	+0	-	-	-
vorbis	329	46	137	+286	+192	+2	-	-	-
zlib	10	3	19	+6	+1	+1	-	-	-

TABLE III: Results benchmark #1c: Symbolic Exploration

The internal structure of *freetype* and *harfbuzz* seems to heavily rely on symbolic pointers. Activating symbolic reads brings 105 and 41 new edges, therefore, a respective gain of 18% and 8%. For *harfbuzz*, symbolic writes also bring an additional 3% gain. It is not improving the coverage of any other target despite generating an extra 283 inputs (+264%). This phenomenon also happens for symbolic reads on *cyclone*, *libjpeg*, *libpng* or *vorbis* where many additional inputs are generated for a very marginal coverage gain. Especially on *harfbuzz*, it increases the corpus by 2772% for a low coverage improvement. To improve scaling in the context of ensemble fuzzing, we decided to disable symbolic pointer enumeration.

VI. Benchmark #2: Ensemble Fuzzing

A. Experimental setup

An ensemble fuzzing campaign is run on the 8 targets for a duration of 24h. The benchmark compares test engines: 1) running independently, 2) with $\overrightarrow{\cup}$ the union of their coverage corresponding to half-duplex mode and 3) $\overrightarrow{\cap}$ the full-duplex mode where fuzzers and TritonDSE share their inputs. Engines are run once on each target. We conducted prior experiments showing that the non-deterministic aspect of greybox fuzzers does not change the relative results between engines. The coverage metric used is edge.

For fairness reasons, all fuzzers run as a single thread and are run without additional dictionary or *cmplog* binary. TritonDSE is run with hyperparameters defined in Section V. It is consequently run with $\mathcal{PE}$ coverage strategy, using Bitwuzla and without symbolic pointer coverage.

B. Coverage Results

Table IV shows coverage reached for each fuzzing campaign for engines alone, the half-duplex ($\overrightarrow{\cup}$) where all engines coverage is aggregated without sharing, and the full-duplex ($\overrightarrow{\cap}$) where inputs are shared between engines. The two ensemble fuzzing modes do not outperform by far, but it does on 6 of the 8 targets. The $\overrightarrow{\cup}$ strategy improves the results for half of the targets which means that while Honggfuzz is accountable for most of the coverage discovery, there are still a few edges only covered by AFL++ or TritonDSE that Honggfuzz did not cover after 24 hours.

Results for $\overrightarrow{\cap}$ outperform $\overrightarrow{\cup}$ on 25% of targets (*libjpeg* and *vorbis*). Counter-intuitively it performs worse than $\overrightarrow{\cup}$, which means that seed sharing has a weakening impact.

	AFL++	Honggfuzz	TritonDSE	$\overrightarrow{\cup}$	full-duplex ($\overrightarrow{\cap}$)	
	AFL	HF	TT	cov	cov	incr-$\overrightarrow{\cup}$
cyclone	1249	1541	1149	**1546**	1544	-2
freetype	3703	12946	3305	**13046**	12865	-181
harfbuzz	4083	**7773**	3702	**7773**	7678	-95
libjpeg	1588	1944	841	1945	**2180**	+237
libpng	797	1005	432	**1016**	978	-38
openthread	1693	2084	1095	**2097**	1963	-134
vorbis	1480	1593	1022	1594	**1596**	+2
zlib	537	**541**	87	**541**	534	-7

TABLE IV: Coverage comparison, fuzzers and half/full-duplex

While the final coverage gives an indication on the state of coverage after 24 hours, the temporal evolution of the coverage is even more important, especially for short-term campaigns of a few hours. Figures 2 gives the coverage evolution for engines alone and half/full-duplex. Results are in logscale to better see the beginning of a campaign.

For *cyclone*, as inputs are rather complex with multiple interdependent fields, it leaves TritonDSE with many opportunities to solve uncovered branches. These new inputs strongly drive coverage improvement in the early hours of the campaign and lead to outperform $\overrightarrow{\cup}$. In this use-case, the `Pastis` full-duplex collaboration helps. Honggfuzz alone only catches up with the coverage just before the 24 hours limit.

For *freetype*, *harfbuzz*, *libpng* and *vorbis*, TritonDSE provides very few inputs and is consequently unhelpful. The main reason is that targets are larger and Honggfuzz generates numerous inputs, thus the replay and emulation takes more time. In the meantime, edges are covered and there are fewer remaining to be solved. Counter-intuitively $\overrightarrow{\cup}$ outperform $\overrightarrow{\cap}$ and slowly catches up with its coverage. This result indicates that introducing new inputs may be pernicious in some cases.

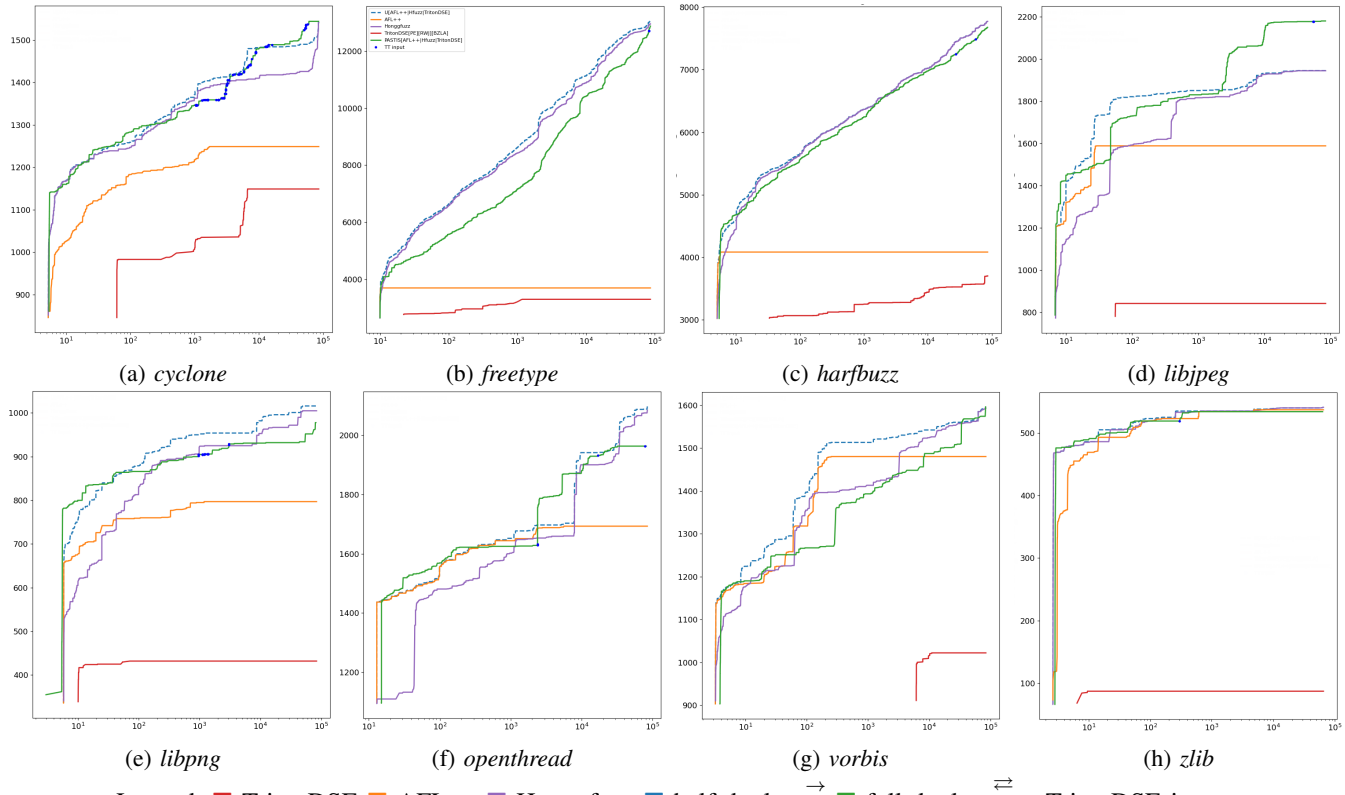
For *libjpeg* in Figure 2d, $\overrightarrow{\cap}$ strongly outperforms $\overrightarrow{\cup}$ by covering 237 more edges. Interestingly TritonDSE does not take action here. The gap is then filled by a great osmosis between Honggfuzz and AFL++ that help one another.

While *zlib* provides no insightful results because it gets completely covered instantly, *openthread* exhibits an interesting behavior. While $\overrightarrow{\cap}$ ultimately loses over $\overrightarrow{\cup}$ at the end of the campaign, it originally outperforms it thanks to TritonDSE that provides a new input that unlock fuzzers to discover significantly more coverage.

These results show that on fuzzbench targets, DSE seldomly helps the overall coverage. The main reason is that Honggfuzz generates a large input corpora that TritonDSE needs to replay to decide whether to keep them or not. Because of the design discussed in Section III this delays the exploration and SMT solving significantly. Also, TritonDSE alone performs insufficiently as it lacks some symbolic modeling of multiple functions used by fuzzbench targets and ultimately loses symbolic tracking.

C. Full-duplex $\overrightarrow{\cap}$ Detailed Results

Previous results show how `Pastis` in $\overrightarrow{\cap}$ mode performs against engines used alone or against simple corpus aggregation $\overrightarrow{\cup}$. It is needed to breakdown the contributions of individual engines within the collaborative mode $\overrightarrow{\cap}$. Table V shows the amount of inputs submitted by each engine during the collaborative campaign. As results show, Honggfuzz accounts for most of the inputs submitted. TritonDSE submits very few inputs for the aforementioned reasons, but as it first replays received inputs to update its own coverage, SMT queries solved necessarily correspond to edges not covered by any other fuzzer.



(a) *cyclone* (b) *freetype* (c) *harfbuzz* (d) *libjpeg*

(e) *libpng* (f) *openthread* (g) *vorbis* (h) *zlib*

Legend: TritonDSE AFL++ Honggfuzz half-duplex $\overrightarrow{\cup}$ full-duplex $\overrightarrow{\cap}$ TritonDSE inputs

Fig. 2: Coverage evolution (logscale)

	seeds	AFL++	Honggfuzz	TritonDSE	Total
cyclone	1	26	1880	182	2088
freetype	2	738	15717	3	16458
harfbuzz	58	601	9253	2	9856
libjpeg	1	405	2615	1	3023
libpng	1	362	856	24	1242
openthread	1	327	834	4	1165
vorbis	1	441	767	0	1208
zlib	1	307	476	1	784

TABLE V: Inputs submission in full-duplex mode

	seeds	AFL++	Honggfuzz	TritonDSE	Total
cyclone	844	3	656	41	1544
freetype	2631	1052	9178	4	12865
harfbuzz	3017	937	3724	0	7678
libjpeg	772	544	863	1	2180
libpng	335	281	352	10	978
openthread	1095	437	425	6	1963
vorbis	883	214	499	0	1596
zlib	66	81	387	0	534

TABLE VI: Engines coverage contribution in full-duplex mode

Coverage: When running alone TritonDSE generates respectively 136 and 169 inputs on *freetype* and *harfbuzz*. As Table V shows in full-duplex it only solves 3 and 2 branches which means other branches were already covered by other fuzzers. This shows that many unnecessary SMT queries are avoided as they got covered by greybox fuzzers. On *openthread*, TritonDSE is able to submit 4 inputs thus negating 4 branches, while when running alone, none of the reachable branches were satisfiable. It implies that in this case fuzzers unlocked new coverage for the DSE.

As input files do not directly relate to coverage, Figure VI shows the direct contributions of each engines to the total edge coverage. Most edges are covered by the seed corpus. The different columns show the additional coverage found by each tool, in other words, the number of edges they were the first to discover.

As expected the initial corpus contributes to most of the coverage. We can compare additional edges discovered by fuzzers. Honggfuzz is the first to discover most edges, which is consistent with previous observations on the number of inputs generated by each engine. However, many of its inputs do not cover any new edges. As an example on *openthread*, AFL++ and Honggfuzz generate respectively 327 and 834 inputs (2.5 times more) but they respectively discover 437 and 425 new edges. As such, AFL++ has better input to coverage ratio. TritonDSE is meant to have 1/1 or more as any solved conditional branch should provide an input that exercises a new edge. However, results of coverage show a slightly inferior ratio. This might be due to an input submission race condition where another fuzzer discovers the same branch and submits its input in the time it takes TritonDSE to perform the DSE and query the solver.

For *openthread*, TritonDSE generates 4 inputs discovering 6 edges. This result does not explain coverage inflation shown in Figure 2f. The explanation is that even though the inputs did not directly cover many new edges, it enabled one of the fuzzers to discover way more coverage.

VII. Related Work

Greybox fuzzing and DSE have widely been studied [1], [15]. Fuzzing is a very active research domain and algorithms are getting more and more sophisticated [16], [17]. However efficiently combining greybox fuzzing and whitebox fuzzing is less studied. The two main approaches are *hybrid fuzzing* [18], where engines are tightly coupled, or *ensemble fuzzing*, where all engines are considered equal from a functional aspect.

Hybrid Fuzzing: The *selective symbolic execution* approach proposed by driller [19] aims to augment AFL with concolic execution (DSE) to only explore paths deemed interesting by the fuzzer that it did not manage to cover. The difficulty with this approach is determining interesting paths and determining when the fuzzer is stuck to launch the symbolic exploration. Still, `Pastis` applies in essence *selective symbolic execution* as it only solves uncovered edges. The main difference is that the driller fuzzer launches the DSE on purpose when blocked while in `Pastis` they run simultaneously all the time.

Another approach is Qsym [20] that deeply intertwines the fuzzing and the concolic engine to improve scaling. They also relax soundness by solving partial formulas and ignore some basic blocks. To further improve performances, the authors get rid of intermediate representation and have a direct translation of instructions to symbolic expressions. This design requires significant modifications of the fuzzer and requires integrating with a single fuzzer. The `Pastis` design choices favor multiples engines and a light integration in each engine.

Ensemble Fuzzing: do not combine engines tightly but consider them equally and emphasizes external mechanisms to synchronize them making them work together. Multiple frameworks have been suggested like deepstate [21], EnFuzz [22] or Collabfuzz [23]. The latter aims at implementing policies at broker level to perform some input filtering or routing to the different engines. Same authors later proposed Cupid [24] to automatically select the right set of fuzzers for collaborative fuzzing. These approaches mostly integrate greybox fuzzers and thus do not address greybox/whitebox interactions. Google, and Microsoft released respectively ClusterFuzz [25] and OneFuzz [26], their fuzzing infrastructures enabling running campaigns using various fuzzers. These frameworks are mature to be used in production, but require a heavy setup and installation process. ClusterFuzz is the underlying infrastructure behind OSS-Fuzz [27].

Recent modular fuzzing framework like LibAFL [28] or centipede [29] are providing built-in mechanisms for collaborative fuzzing. As such, LibAFL defines LLMP[7] (Low Level Message Passing) for client synchronization using a broker mechanism. Primarily designed for inter-process communication it also enables communication over the network.

[7] https://aflplus.plus/libafl-book/message_passing/message_passing.html

VIII. Discussion

Limitations: In a collaborative environment, using coverage provides the direct contribution of each engine to the overall results, but indirectly represents help provided by engines to each other. Gathering this information would require a better way to instrument fuzzers to study the beneficial or prejudicial impact of inputs received.

Although the DSE has been tuned to better scale (cf. Benchmarks #1) it is still orders of magnitude slower than greybox fuzzing. Combined with high input throughput fuzzers like Honggfuzz consequently inhibit DSE's effectiveness. Side experiments performed with solely TritonDSE and AFL++ have shown that it helps AFL++ more significantly. As results show, even used alone, TritonDSE covers way less edges than fuzzers. The reason is, the lack of symbolic modeling for many libc functions and syscalls.

Threat to Validity: Each benchmark was launched once, thus it can imply a statistical bias induced by fuzzers non-determinism. Also, for Honggfuzz, inputs received are properly introduced in his queue, but we have no feedback on whether it has properly been executed afterward nor if it was a valuable input.

Future Work: On the DSE side, efficiency can be improved by relaxing soundness assumption like Qsym or under-constrained symbolic execution approaches [30]. TritonDSE might also benefit from running in a concolic manner directly within a QBDI instrumentation for instance.

On the fuzzing side, fuzzers like AFL++ run with simple configuration. One can study the impact of *cmplog* or dictionaries on the ensemble fuzzing campaign. As results show for many targets, the coverage is still evolving when the 24h deadline is triggered. An axis of research is running longer campaigns to evaluate if $\overrightarrow{\cap}$ outperforms $\overrightarrow{\cup}$ in the long run. If so, evaluating precisely the lagging induced by the seed sharing would be valuable.

Also, only naive policies have been tested, half-duplex $\overrightarrow{\cup}$, and full-duplex $\overrightarrow{\cap}$. Notwithstanding Collabfuzz research [23] suggests that no clear policy is outperforming, it is interesting to test unidirectional approaches based on the engine type, like whitebox→greybox only, or the reverse. As many inputs shared do not improve coverage it would be valuable to filter inputs strictly generating new coverage at broker level. However, that implies the broker playing an active replay role instead of just being a dispatcher.

`Pastis` has been designed with closed-source scenarios in mind, thus, it was not appropriate using Fuzzbench [12] tooling that relies for instance, on instrumentation from source for the coverage. It was also not possible to monitor the ensemble fuzzing collaboration as precisely as being done in `Pastis` broker. Nonetheless, integrating `Pastis` in fuzzbench enables, on open-source targets, to relate results with the other engines supported.

IX. Conclusion

This paper shows experimental results of a collaborative ensemble fuzzing framework combining greybox and whitebox fuzzing. It brings new token of appreciation for **RQ1** showing that DSE can help greybox fuzzing achieving better coverage on some targets. These early results give another evidence that ensemble fuzzing and especially $\overrightarrow{\cup}$ is outperforming fuzzers alone especially at the beginning of a campaign. Indeed, fuzzers are certainly initially exploring different part of the program and then converging on harder ones. Thus ensemble fuzzing is beneficial in short-term campaigns which makes it suitable to be used in CI/CD context. For longer campaigns, these results show that Honggfuzz used alone provides a better cost-to-benefit trade-off in terms of computational time.

Yet, results show that there is room for improvement for $\overrightarrow{\cap}$ in order to give a positive answer to **RQ2**. Undeniably, Honggfuzz is noisy in terms of input generation which combined with TritonDSE inhibits its effectiveness. However these, results open the way to explore more efficient collaboration strategies or seed sharing schemes. The two open-source frameworks `Pastis` and TritonDSE provide a flexible experimental framework for further testing.

References

[1] V. M. Manes, H. Han, C. Han, S. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, nov 2021.

[2] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, "A systematic review of fuzzing techniques," *Computers & Security*, vol. 75, pp. 118–137, 2018.

[3] T. Wang, T. Wei, G. Gu, and W. Zou, "Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution," *ACM Trans. Inf. Syst. Secur.*, vol. 14, no. 2, sep 2011.

[4] 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, 2016, p. 1032–1043.

[5] P. Godefroid, M. Y. Levin, and D. A. Molnar, "SAGE: whitebox fuzzing for security testing," *Commun. ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[6] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008*, 2008, pp. 209–224.

[7] 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, June 3-5 2015*. SSTIC, 2015, pp. 31–54.

[8] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna, "SOK: (state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, 2016, pp. 138–157.

[9] T. of Bits, "Maat - open-source symbolic execution framework," 2022.

[10] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, "Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, Sep. 2019, pp. 1–15. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/wang

[11] C. Hubain and C. Tessier, "Implementing an llvm based dynamic binary instrumentation framework," 2017.

[12] D. Asprone, J. Metzman, A. Arya, G. Guizzo, and F. Sarro, "Comparing fuzzers on a level playing field with fuzzbench," in *15th IEEE Conference on Software Testing, Verification and Validation, ICST 2022, Valencia, Spain, April 4-14, 2022*. IEEE, 2022, pp. 302–311.

[13] L. M. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, 2008, pp. 337–340.

[14] A. Niemetz and M. Preiner, "Bitwuzla at the SMT-COMP 2020," *CoRR*, vol. abs/2006.01621, 2020. [Online]. Available: https://arxiv.org/abs/2006.01621

[15] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 50:1–50:39, 2018.

[16] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "BEACON: directed grey-box fuzzing with provable path pruning," in *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, pp. 36–50.

[17] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, "Stateful greybox fuzzing," in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022, pp. 3255–3272.

[18] B. Pak, "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution," 2012.

[19] 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 *23rd Annual Network and Distributed System Security Symposium, NDSS*, 2016.

[20] 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)*. USENIX Association, 2018, pp. 745–761.

[21] P. Goodman, G. Grieco, and A. Groce, "Tutorial: Deepstate: Bringing vulnerability detection tools into the development cycle," in *2018 IEEE Cybersecurity Development, SecDev 2018, Cambridge, MA, USA, September 30 - October 2, 2018*, 2018, pp. 130–131.

[22] 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, Santa Clara, CA, USA, 2019*. USENIX Association, 2019, pp. 1967–1983, `[site]`.

[23] S. Österlund, E. Geretto, A. Jemmett, E. Güler, P. Görz, T. Holz, C. Giuffrida, and H. Bos, "Collabfuzz: A framework for collaborative fuzzing," in *Proceedings of the 14th European Workshop on Systems Security*, ser. EuroSec '21, 2021, p. 1–7.

[24] E. Güler, P. Görz, E. Geretto, A. Jemmett, S. Österlund, H. Bos, C. Giuffrida, and T. Holz, "Cupid : Automatic fuzzer selection for collaborative fuzzing," in *Annual Computer Security Applications Conference*, ser. ACSAC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 360–372.

[25] Google, "Clusterfuzz - scalable fuzzing infrastructure," 2021.

[26] Microsoft, "Onefuzz - a self-hosted fuzzing-as-a-service platform," 2021.

[27] Google, "Oss-fuzz - continuous fuzzing for open source software."

[28] A. Fioraldi, D. Maier, D. Zhang, and D. Balzarotti, "LibAFL: A Framework to Build Modular and Reusable Fuzzers," in *Proceedings of the 29th ACM conference on Computer and communications security (CCS)*, ser. CCS '22. ACM, November 2022.

[29] Google, "Centipede - a distributed fuzzing engine," 2022, `[code]`.

[30] D. A. Ramos and D. R. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, 2015, pp. 49–64.