FishFuzz: Throwing Larger Nets to Catch Deeper Bugs

Han Zheng*, Jiayuan Zhang†*, Yuhang Huang*, Zezhong Ren*, He Wang†
Chunjie Cao‡, Yuqing Zhang*§, Flavio Toffalini†, Mathias Payer†

* National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences, China
† School of Computer and Communication, Lanzhou University of Technology, China
‡ School of Cyber Engineering, Xidian University, China
§ School of Cyberspace Security, Hainan University, China
† EPFL, Switzerland

Abstract—Greybox fuzzing is the de-facto standard to discover bugs during development. Fuzzers execute many inputs to maximize the amount of reached code. Recently, Directed Greybox Fuzzers (DGFs) propose an alternative strategy that goes beyond “just” coverage: driving testing toward specific code targets by selecting “closer” seeds. DGFs go through different phases: exploration (i.e., reaching interesting locations) and exploitation (i.e., triggering bugs). In practice, DGFs leverage coverage to directly measure exploration, while exploitation is, at best, measured indirectly by alternating between different targets. Specifically, we observe two limitations in existing DGFs: (i) they lack precision in their distance metric, i.e., averaging multiple paths and targets into a single score (to decide which seeds to prioritize), and (ii) they assign energy to seeds in a round-robin fashion without adjusting the priority of the targets (exhaustively explored targets should be dropped).

We propose FishFuzz, which draws inspiration from trawl fishing: first casting a wide net, scraping for high coverage, then slowly pulling it in to maximize the harvest. The core of our fuzzer is a novel seed selection strategy that builds on two concepts: (i) a novel multi-distance metric whose precision is independent of the number of targets, and (ii) a dynamic target ranking to automatically discard exhausted targets. This strategy allows FishFuzz to seamlessly scale to tens of thousands of targets and dynamically alternate between exploration and exploitation phases. We evaluate FishFuzz by leveraging all sanitizer labels as targets. Extensively comparing FishFuzz against modern DGFs and coverage-guided fuzzers shows that FishFuzz reached higher coverage compared to the direct competitors, reproduces existing bugs (70.2% faster), and finally discovers 26 new bugs (18 CVEs) in 44 programs.

I. INTRODUCTION

Greybox fuzzing is the established technique to automatically test and find bugs in programs. The base concept of a fuzzer is as simple as effective: execute the target program with inputs (seeds), observe its behavior, and report observed crashes. The seed that triggered the crash allows reproducing the crash later during debugging. The effectiveness of these fuzzers moved researcher and companies to invest considerably effort in this technology, thus producing more sophisticated fuzzing designs [1]–[10].

Greybox fuzzing is coverage-guided, which means that fuzzers aim at maximizing the amount of explored code [1], [2], [9]. However, solely increasing coverage does not guarantee finding all bugs: once code is reached (covered), the fuzzer drives inputs toward unexplored regions, thus missing the opportunity to trigger potential error cases in already code. Fuzzing encompasses two aspects: exploration and exploitation. A fuzzer needs both to explore the program broadly but also trigger bugs in code it reaches. During exploration, the main goal is to increase coverage. During exploitation, the main goal is to trigger bugs by executing a piece of code with diverse inputs. Greybox fuzzers directly measure exploration by tracking newly reached code areas but have no feedback for exploitation. Greybox fuzzers assume that exploitation is implicitly covered by random mutations that repeatedly execute the same code accidentally.

To make exploitation a first class citizen, researchers introduced Directed Greybox Fuzzers (DGF [11]) which direct exploration towards a specific code location (target), improving the likelihood to find bugs at that location [3], [7], [8], [11]. DGFs leverage the distance between seeds and targets to fine-tune inputs more likely to trigger errors.

Recent DGFs try to balance exploitation and exploration to automatically trigger larger sets of targets (on the order of thousands) [3], [7], [8], [12]. However, we observe two core limitations in previous works. First, they model the distance between seeds and targets as a single harmonic average. This means that, regardless of the size of the target set, they always collapse the distance seeds-targets into a single scalar. In short, it does not matter what target is reached as long as any target is reached. This approach intrinsically limits the precision of the fuzzer since, with larger target sets, the distance tries to fit (unrelated) targets contemporaneously. Moreover, current distance metrics are affected by unresolved indirect jumps. Second, current work assigns a time-invariant priority to the targets. They use different techniques (e.g., static analysis, sanitizer labels) to infer error-prone targets ahead of time, i.e., before the beginning of the fuzzing session. However, we observe that priorities of a target change during the fuzzing campaign and a fuzzer must adjust its strategy accordingly. For instance, targets that have been hit frequently are “well explored” and therefore less likely to be buggy. Less explored targets should therefore be prioritized to uncover new bugs. Therefore, a time-invariant priority might misclassify the importance of a code location and waste fuzzing energy.
Observing the imprecision of current distance metrics and time-invariant target priority, we propose FishFuzz: a novel DGF that seamlessly scales to tens of thousands of targets automatically. FishFuzz builds on two key contributions: (1) our novel multi-distance metric whose precision is independent of the number of targets and more robust to indirect jumps, and (2) a dynamic target ranking to automatically discard exhausted targets and steer fuzzer energy towards (more) promising locations. The insight behind our approach is analogous to trawling (which inspired our fuzzer’s name). After casting a wide net (capturing many possible targets), the net is closed gradually. During exploration our fuzzer tries to reach as many targets as possible while during exploitation, our fuzzer tracks how well explored each target is.

We use our insights to design a novel multi-stage seed selection strategy that promotes seeds according to the current phase the fuzzer is in: exploration or exploitation. Unlike previous works, FishFuzz can easily handle tens of thousands of targets without loss of precision (e.g., in our largest target, we cover over 20k targets). Concurrently, FishFuzz automatically discards unfruitful targets and priorities rarely tested ones. The combination of these two strategies allows the fuzzer to reach more targets (during exploration) and then spread its energy evenly across the discovered targets (during exploitation). Together, this results in better bug finding capabilities.

Our FishFuzz prototype extends AFL and we compare it against three modern DGFs (i.e., ParmeSan [3], TortoiseFuzz [5], and SAVIOR [8]) as well as against AFL++ [2] and AFL [1]. We conduct experiments over three benchmarks. The first two are composed of 9 and 7 programs taken from TortoiseFuzz [5] and SAVIOR [8], respectively. For the third one, we select 28 real programs from other top tier fuzzing works. For what concerns exploration, our evaluation shows we easily reach an higher coverage (up to 132% more) and hit more targets (up to 116% more) compared with the state of the art. In terms of exploitation, we show FishFuzz can trigger 62% more targets and find up to 2x unique bugs with respect to its competitors. More precisely, FishFuzz easily reproduces 45 previous bugs, among which 33 in less time (70.2%) compared to previous works. Moreover, we discover 25 new bugs from which 18 are already confirmed CVEs. Additionally, we measure the contribution of exploration and exploitation phases. The results show (1) FishFuzz better balances the fuzzing energy among the targets and (2) we show the impact of the exploration and exploitation strategy in terms of coverage and targets trigger.

To sum up, our contributions are:

- A detailed evaluation against the state of the art and new 25 bugs found (18 CVEs) in 44 programs.
- A dynamic target ranking that automatically guides fuzzer energy towards promising locations, while discarding thoroughly explored ones.
- A novel multi-distance metric between seeds and targets that is independent of the size of the target set and robust against unsolved indirect jumps.
- A detailed evaluation against the state of the art and new 25 bugs found (18 CVEs) in 44 programs.

We will release the full source code with the publication of the paper and a demonstration prototype is available at https://zenodo.org/record/6405418

II. BACKGROUND

FishFuzz heavily modifies seed selection and introduces a new distance metric. To understand the limitations of existing queue culling and distance metrics, we first introduce the concepts and then highlight challenges in their current form.

A. Queue Culling

Modern fuzzers, such as AFL [1] and AFL++ [2], take as input a program and a set of inputs (seed) to submit to the program. Their workflow is loop-based: they select seeds, mutated them, and submit them to the program. The fuzzer collects information about the program execution (e.g., code coverage) to guide the next fuzz iteration. Usually, fuzzers select seeds to improve the code coverage (code-coverage guided fuzzers), but this behavior can be adjusted with different metrics and purposes.

To select interesting seeds, fuzzers adopt two strategies: input filtering and queue culling. With input filtering, we refer to strategies that discard unproductive seeds, while queue culling gives more priority to interesting seeds (without discarding others).

Our work focuses on queue culling strategies. Specifically, these approaches use a specific flag, called favor, to indicate whose seeds will be selected in the next fuzz iteration. The favor setting can follow vary strategies according to the results one wants to obtain. For instance, we can select seeds to improve the coverage, or else we can guide the testing toward specific code locations in the attempt to trigger a specific bug.

In case of AFL [1], it maintains a map (top-rate) that pairs visited edges and the best input for visiting it, where the best input is simply the smallest and fastest to reach that edge. While exercising inputs, AFL traces the visited edges and if the input results better (i.e., faster/smaller to reach it) for some edges in top-rate, than AFL assigns the new the input to those edges. In this way, AFL can easily find suitable inputs for an edge through a fast look-up. More advanced fuzzers, such as Angora [6] or AFL-Sensitive [13], include additional information, such as the calling stack, the memory access address, and the n-basic block execution path. More recent DGFs, such as TortoiseFuzz [5], infer the best seed based on a combination of static analysis and seed’s execution path, e.g., if the seed has probability to hit sensitive code locations. Regardless the complexity of the current cull queue algorithms, we observe a main limitation: they do not consider dynamic information about the targets status. For instance, if a portion of code has been triggered, the cull queue should stop considering that area as interesting and select seeds that hit other locations. Conversely, current cull queue algorithms estimate the targets priority ahead-of-time, without reconsidering them during the campaign. To tackle
such problems, FishFuzz uses a novel queue culling that employs a dynamic target ranking that mutates during the fuzzing campaign (Section III-B).

B. Directed Greybox Fuzzers

Unlike traditional greybox fuzzers, which optimize for maximum code coverage, DFG tries to reach specific code locations [11]. To achieve this goal, DFGs define distance-based mechanisms that direct the fuzzers towards the targets via gradually reducing the distance [3], [7], [11]. However, traditional solutions only calculate one harmonic distance between seed and a set of targets, which try to cover all the target via one distance and cannot scale to large number of targets. Conversely, FishFuzz proposes a dynamic multipledistance measurement that estimates the distance between the seeds and each target, respectively. Our approach further expands the scope of the DGFs and scales them to large-scale targets fuzzing.

To show the limitations of current DGFs, we rely on the example in Figure 1. In this scenario, we assume having a call-graph in which we exercise two seeds, $s_1$ and $s_2$, whose execution paths are red and green colored, respectively. Moreover, we assume having a target set $T$ composed of $t_1$, $t_2$, and $t_3$. Finally, we consider the graph’s edges with an uniformly weighted as one. Without loss of generality, we apply to the graph a simplified version of the harmonic-average distance used in AFLGo [11] and ParmeSan [3]. The estimation is done in two steps. First, we compute the distance of each seed $s$ against each target $t$ as the minimum number of edges between the execution path of $s$ and the target $t$, i.e., the distance between $s_1$ and $t_1$ is one (the edge $f_c-f_d$). Then, we compute the harmonic-average among all the distances between the seeds and the targets $T$, that result in $s_1 = 1.8$ and $s_2 = 2.25$. Having this estimation, the fuzzer chooses $s_1$, thus privileging $t_1$. The consequence is that the fuzzer becomes biased against $t_1$ and misjudge $t_2$ and $t_3$. Since $t_2$ and $t_3$ fall far from $t_1$, the fuzzer hardly mutates seeds in that direction. Even worse, in case $t_1$ is triggered, the fuzzer will keep hitting $t_1$ since the distances only accounts graph’s static information. In FishFuzz, we propose a novel multi-distance measure that mitigates such problems.

III. FishFuzz Design

Directed fuzzers are hampered by two key problems:

- **P1: Seed explosion.** The fuzzing process produces a large amount of seeds that are all selected in round robin fashion. This results in promising seeds not receiving sufficient attention or, even worse, no fuzzing cycles at all.

- **P2: Imprecise distance estimation.** Calculating distance to many targets (e.g., all sanitizer labels) introduces imprecision and overwhelms the selection prioritization algorithms of existing DGFs.

FishFuzz introduces a smart seed selection strategy combined with a novel distance metric to address P1 and P2. Specifically, we implement the seed selection as a queue culling algorithm that assigns the seed priority at each fuzzing iteration (more details in Section II-A). Our approach allows DGFs to handle programs with a large number of targets.

FishFuzz dynamically adapts the priorities of the targets to privilege unexplored ones—they that have not yet been covered and which are more likely to contain unseen bugs, reducing the energy for those targets that were already sufficiently explored. Moreover, FishFuzz models the distance from seed to targets as a multi-distance function that overcomes the limitations of previous works (Section II-B). Such mechanism works at function level and combine light static analyses with dynamic information from the fuzzing session.

Our design is the result of two key limitations observed in the distance estimation of previous works. First, existing DGFs measure the distance between seeds and targets as a single harmonic average. This approach synthesizes the information of all the targets into a scalar [3], [11] (Section II-B). While this works for a small set of few targets, it intuitively loses precision when their number increases. For instance, the first work on DGF dealt with tens of targets [11], while FishFuzz easily scales to tens of thousands targets. In other words, modeling the seed-targets distance with a single harmonic average is equivalent to hit multiple targets with a single seed. Secondly, all previous DGFs consider the targets as time-invariant, which means the target importance is statically assigned before the campaign and cannot change. This intuitively dissipates fuzzing energy because we might end up hitting unfruitful targets while overlooking more promising ones. For instance, intensively tested targets are less likely to reveal new bugs, while poorly explored targets can still express errors [3], [4]. All these considerations drive the FishFuzz seed selection strategy, as we explain in the following sections.

The design of FishFuzz extends classic greybox fuzzers [1]. Our changes focus on a new queue culling algorithm and a set of auxiliary structures to retain dynamic target information and distance metrics. Our design allows us to improve the seed selection based on extra information not available otherwise. The overall workflow is depicted in Figure 2 and recalls standard fuzzing procedures [1–3], [4].

Given a target program, we compile and instrument it with
specific sanitizers (①), this phase makes the instrumented program suitable for the fuzzing campaign (②), and extracts initial information useful for FishFuzz (③). We detail the latter in Section III-A. The instrumented program follows the standard greybox workflow in which a fork server handles the program lifecycle (④). Meanwhile, a fuzz loop selects inputs from a queue and submits them to the program instances (⑤). The input selection is handled by a queue culling algorithm (⑥) that relies on our novel distance metric (⑦) and a dynamic target ranking (⑧). We detail the queue culling and the distance metric in Section III-B and Section IV, respectively. The target ranking is a shared structure that tracks meta information about the targets, e.g., hit frequency or if a target has been reached by a seed. We use this information in the distance calculation and the cull queue.

A. Program Preparation

In the program analysis phase, we compile the program and generate a fuzzing compatible binary. We also instrument the code with extra components for code coverage and security sanitizers similarly to previous works [1], [3], [5], [6], [11]. The FishFuzz design is agnostic by the sanitizer used, in our experiment we successfully tested ASan [14], [15] and UBSan [16]. We use the sanitizer information to extract the program targets. Finally, we perform a lightweight static analysis at compilation time.

Target Extraction: Our fuzzer uses sanitizer check locations as target labels. We rely on off-the-shelf sanitizers to extract targets from the program. Specifically, we locate the sanity checks injected on top of the original program and consider them as targets to explore. The intuition is that any sanity checks can potentially reveal the presence of a bug. Our fuzzer then explores towards these code locations in the attempt to trigger (exploit) them. FishFuzz is agnostic to the nature of the targets as long as they are uniquely identifiable in the program. As studied previously [3], [4], some targets might be unreachable at runtime (e.g., due to system environment). Current approaches use static analysis to discard these cases, however, as the same authors claim, using solely static analysis risks to also remove correct sanity checks. Conversely, FishFuzz initially considers all the targets as valid and it deals with false positive by dynamically ranking the targets and filtering out unpromising ones. Our approach removes possible errors from unsound static analysis (details in Section III-B).

Static Analysis: During compilation, we extract the control-flow-graph (CFG) and the call-graph (CG). This initial analysis, for now, is oblivious to indirect calls. Then, FishFuzz relies on CFG and CG in a novel inter-function distance to select seeds closer to a given target. We perform this operation at LLVM-IR [17]. We describe the full function distances algorithm in Section IV-A and discus how it deals with indirect calls in Section IV-D.

B. Queue Culling Algorithm

We design the queue culling by taking inspiration from the trawl fishing technique. At the beginning of the fuzzing campaign, FishFuzz prioritizes the function exploration (expanding the net). When no new functions are reached, FishFuzz focuses on maximizing the reached targets (the net starts closing). Once enough targets are reached, the cull logic changes.
again and tries to trigger the interesting targets (catching as many fish as possible). Since every phase requires different metrics (i.e., number of functions or targets reached/triggered), we adopt a multiple phases approach as suggested from previous works [3]. Specifically, FishFuzz relies on three phases: inter-function exploration, intra-function exploitation, and exploitation – all pictured Figure 3. The purpose of the inter-function exploration is to reach interesting functions and it leverages our novel multi function-level distance (i.e., expanding the net). The intra-function exploration, instead, focuses on the internal function testing, relies on the standard AFL mutation algorithm, and tries to hit as many targets as possible (i.e., start closing the net). Finally, the exploitation phase drives the fuzzer energy to trigger the maximum number of reachable targets (i.e., catching the fish). This phase uses a dynamic target ranking to prioritize promising locations.

The switch among the different phases happens at specific events: (i) every time new function is traversed, (ii) if no new function is found for a period of time (i.e., 30min), (iii) if no new target is reached for a period of time (i.e., 10min), and (iv) if no new target is triggered for a period of time (i.e., 1hour). In our experiments, we determined these timeouts for each event and leave further per-target tuning as future work.

Our algorithm overcomes two shortcomings that affected previous works. First, the inter-function exploration uses a novel function distance that is faster in selecting seeds closer to targets. This improves the slow initial phase that affected previous fuzzers [3], [5]. Second, we boost the exploitation phase with a multi-distance function and a dynamic target ranking. Our approach discards non-profitable targets, thus it leverages our novel multi function-level distance (i.e., expanding the net). The intra-function exploration, instead, focuses on the internal function testing, relies on the standard AFL mutation algorithm, and tries to hit as many targets as possible (i.e., start closing the net). Finally, the exploitation phase drives the fuzzer energy to trigger the maximum number of reachable targets (i.e., catching the fish). This phase uses a dynamic target ranking to prioritize promising locations.

In the rest of this section, we detail the inter-function exploration and the exploitation phase. For what concerns the intra-function exploration, we used the standards AFL cull algorithm [1], we thus omit its description for simplicity.

**Inter-function Exploration Phase:** In this phase, FishFuzz selects seeds to maximize the reached functions containing targets. The cull algorithm of this phase is shown in Algorithm 1. Specifically, given a Queue of seeds and a set of Functions from the target program, FishFuzz sets favor = 1 to the closest seed for each unexplored function that also contains targets (line 6). After the favored seeds are submitted to the program, FishFuzz updates the list of explored Functions and repeat the process. getClosestSeedToFun finds the closest seed s to the function f through a seed-function distance that we discuss in Section IV-A in case of multiple seeds equally distant to f, we prefer the lowest execution time.

**Exploitation Phase:** In the exploitation phase, FishFuzz tries to trigger the maximum number of targets previously reached. Our intuition is to keep hitting the same target with different seeds (that can reach the target), thus increasing the chance to expose a bug. Algorithm 2 shows the pseudo-code of this phase. Specifically, we first select those targets that are reached through either the inter- or the intra-function exploration phase (line 6 to line 10). Among the trgs_to_visit, we select the top 20% of lesser hit targets (line 12 and line 14). For each suitable target, getFastestSeedToTarget returns the fastest seed s (with lowest execution time) that hits t, thus it finally sets it as favor. In this phase we have high probability to have seeds that hit targets (i.e., seed-target distance = 0) due the exploration phase. Finally, the function getFastestSeedToTarget relies on a seed-target multi-distance function that we detail in Section IV. In our prototype, we considered the top 20% of lesser hit targets, we leave the study of optimal threshold values as future work.

Our approach overcomes two important limitations of previous DGFs [3], [5]. First, FishFuzz has a dynamic view of the target importance, i.e., a triggered target looses importance in the campaign, while less tested targets receive more energy. Second, we automatically discard unreachable targets without employing heavy software analysis [3], [5], [7], thus avoiding intrinsic false positives. This philosophy is also reflected in the seed-target distance, as explained in Section IV.

---

**Algorithm 1:** Cull logic for the inter-function exploration phase.

```plaintext
interFunctionCullQueue(Queue, Functions)
for s ∈ Queue do
    s.favor = 0
end
for f ∈ Functions do
    if f.unexplored ∧ f.hastargets then
        s ← getClosestSeedToFun(Queue, f)
        s.favor = 1
    end
end
```

**Algorithm 2:** Cull logic for the exploitation phase.

```plaintext
exploitationCullQueue(Queue, Targets)
for s ∈ Queue do
    s.favor = 0
end
trgs_to_visit ← ∅
for t ∈ Targets do
    if t.reached then
        trgs_to_visit ← trgs_to_visit ∪ {t}
    end
    trgs_to_visit ← orderByHit(trgs_to_visit)
    threshold ← |trgs_to_visit| * 20%
    for (p, t) ∈ enumerate(trgs_to_visit) do
        if p < threshold then
            s ← getFastestSeedToTarget(Queue, t)
            s.favor = 1
        end
    end
```
IV. DISTANCE MEASUREMENT FOR *FishFuzz*

*FishFuzz* uses a novel function distance calculation that improves precision while reducing complexity compared to existing work. Specifically, our solution does not require heavy software analysis to resolve indirect jumps. Informally, this is done by mapping the distance between couple of functions, e.g., \( dff(f_a, f_b) \) indicates the distance between the function \( f_a \) and \( f_b \). *FishFuzz* relies on this approach to either calculate seeds to function as well as seeds to target distance. The algorithm is composed on two steps. At compilation time, we analyze the LLVM-IR code and build a static distance map between functions (Section IV-A). In the fuzzing session, we leverage on the static distance to estimate the distance between seeds and a given function (Section IV-B). We further rely on the static distance to select the closest seeds to a set of targets (Section IV-C). In the last section, we discuss how our function distance deals with indirect calls (Section IV-D).

A. Static Function Distance

Before the start of fuzzing, *FishFuzz* generates a static map containing relationship between functions.

*FishFuzz* first assigns a weight for each function pair \((f_i, f)\) such that \( f \) is a callee of \( f_i \). The weight represents the minimum number of conditional edges that a seed might traverse from the entry point of \( f_i \) to the callee function \( f \), and is computed with the function \( dbb(m_a, m_b) \) (i.e., distance from basic block \( m_a \) to \( m_b \)). Formally speaking, given two functions \( f_i \) and \( f \), we defined \( weight(f_i, f) \) as follow:

\[
weight(f_i, f) = \begin{cases} 
\min dbb(m, m_f) & \text{if } \exists m_f \in f_i \\
\infty & \text{otherwise,}
\end{cases}
\]

(1)

where \( m \) is the first basic block of the function \( f_i \), and \( m_f \) is a basic block belonging to \( f \) and with a function call to function \( f \) (so \( f \) is a callee of \( f_i \)). If \( f \) is a callee of \( f_i \), the weight between \( f_i \) and \( f \) is the minimum distance between \( m \) and \( m_f \). Otherwise, it is unreachable (\( \infty \)). To handle multiple function calls to \( f \), we consider only the minimum distance to leave \( f_i \).

Once the weights are computed, *FishFuzz* defines the distance between two functions as the sum of their weight along the shortest path between two functions by following the CF extracted at compilation time. Formally speaking, the distance between two function \( f_a \) and \( f_b \) is defined as follow:

\[
dff(f_a, f_b) = \sum_{f_i \in pf(f_a, f_b)} weight(f_i, f_{i+1}),
\]

(2)

where the function \( sp(f_a, f_b) \) returns the shortest path between \( f_a \) and \( f_b \) using Dijkstra’s algorithm [18], and \( weight(f_i, f_{i+1}) \) is Equation 1 over two consecutive functions in the path.

B. Dynamic Seed to Function Distance

Having the static distance calculated in Section IV-A, we define a function \( dsf(s, f) \) that represents the distance between the functions traversed by the seed \( s \) and a function \( f \) as follow:

\[
dsf(s, f) = \begin{cases} 
\min_{f_i \in \xi(s), f \notin \xi(s)} dff(f_s, f) & \text{if } f \notin \xi(s) \\
0 & \text{otherwise,}
\end{cases}
\]

(3)

where \( \xi(s) \) is the set of functions traversed by the execution of the seed \( s \). In case the \( s \) already hits \( f \), we consider the distance as zero.

With the dynamic seed distance, *FishFuzz* chooses the closest seed for a target function upon the intuition that seeds closer to a target have higher probability to reach it. We mainly use \( dsf \) in the inter-function exploration (Section III-B).

C. Dynamic Seed to Multi-Target Distance

*FishFuzz* employs a novel multi-target distance to estimate seeds closer to a set of targets \( T_s \). Differently from previous works [3], [5], [11], which represent the seed-targets distance a single harmonic average, *FishFuzz* models the seed-targets distance separately. Precisely, *FishFuzz* defines a function \( D \) as follow:

\[
D(s, T_s) \rightarrow (d_{t_1}, \ldots, d_{t_n}),
\]

(4)

where \( s \) is a seed, \( T_i \) is a set of targets, and \( d_{t_i} \) represents the single distance between \( s \) and the target \( t_i \). The size of the vector \((d_{t_1}, \ldots, d_{t_n})\) is equal to the size of the set \( T_s \).

Each \( d_{t_i} \) is defined as follow:

\[
d_{t_i} = t_i,nottriggered \ast dsf(s, f_{t_i}) \mid t_i \in f_{t_i},
\]

(5)

where \( t_i,nottriggered \) is 1 if \( t_i \) has never been triggered, 0 otherwise. \( dsf(s, f_{t_i}) \) is the function distance between the seed \( s \) and the function \( f_{t_i} \) containing the target \( t_i \) (Equation 3).

This distance is used in the exploitation phase (Section III-B) and has two main advantages: first, it automatically excludes targets already triggered (i.e., \( t_i,nottriggered \)), second, the distance between seed and targets is not affected by the size of \( T_s \).

D. Indirect Call handling

Unresolved indirect calls might affect the quality of the fuzzing campaign. Existing DGFs usually mitigate this issue by resolving the indirect jumps in the CFG and CG. For instance, Hawkeye [7] introduces inclusion-based pointer analysis into CG generation, but this strategy does not cover all the indirect calls. ParmeSan [3] has an ad-hoc fuzzing session to dynamically reconstruct the missing edges in CFG and CG.

In *FishFuzz*, the dynamic seed to function distance (Equation 3) already provides a good approximation of indirect calls without the burden to resolve them. To explain this property, we rely on Figure 4 which shows a CFG where the functions pair \( f_c \cdot f_d \), and \( f_b \cdot f_g \), are connected through an (unresolved) indirect call. This example explains two cases. First, we assume the fuzzers have generated a seed \( s_1 \) that traverses \( f_a, f_c, f_d \), and finally hit \( f_f \) (shown in red). When this occurs, *FishFuzz* has enough information to compute the distance between \( s \) and \( f_c \), that is exactly the distance between \( f_d \) and \( f_c \). Using our approach, we say we have
an approximation of the distance because we cannot estimate the component weight \((f_c, f_d)\) (Equation 2). However, since the distance seed-function is expressed as the minimal function distance, we argue our approach statistically finds a quasi-optimal result. The second case, instead, \(f_g\) has no direct calls to resolve the distance calculation. In this case, we assume the fuzzer generates a seed \(s_2\) that traverses \(f_g\). According to Equation 3, this results in distance zero since \(s_2\) hits the target function. Generalizing the example, we can say that whenever a seed traverses an indirect call, FishFuzz can use nearby (connected) functions to estimate the minimum distance between the seed’s execution path and the target function. In case a function has no direct connections, either the fuzzer generates a seed that reaches the function or the latter is unreachable. In practice, our approach is similar to previous works that use a fuzzing session to explore indirect jumps \(f_g\). However, those works apply such approach only once before fuzzing. On the contrary, FishFuzz benefits of each indirect jump resolved at any time of the session.

V. IMPLEMENTATION

The FishFuzz implementation extends AFL [1] version 2.57b and LLVM [17] version 12.0.1.

We implement the inter-function exploration and the exploitation phases as two cull queue functions in AFL for a total of around 2,500 LoC. For the program analysis, we develop additionally analysis passes for LLVM to extract CFG, CG, and estimate the static function distance. Moreover, we develop an additional instrumentation pass to extract information for the dynamic seed to function metric. The LLVM code is around 1,500 LoC in total. Additionally, we have a few python scripts for the compilation process, which is around 200 LoC. As for sanitizers, we use ASan [14, 15] and UBSan [16] distributed with the compiler-rt libraries from LLVM/Clang.

The source code of FishFuzz, along with the material for replicating the experiments, will be released open-source upon acceptance.

VI. EVALUATION

We evaluate the performance of FishFuzz respect to the state of the art. In particular, we desire to answer to the following research questions:

\[ RQ1: \text{How many targets does FishFuzz reach? (Section VI-A)} \]
\[ RQ2: \text{Does FishFuzz balance the energy (Section VI-B)?} \]
\[ RQ3: \text{How efficiently does FishFuzz find bugs (Section VI-C)?} \]
\[ RQ4: \text{Can FishFuzz find new bugs (Section VI-D)?} \]
\[ RQ5: \text{How does FishFuzz redistribute exploration and exploitation (Section VI-B)?} \]
\[ RQ6: \text{Can other fuzzers benefit from our strategies (Section VI-F)?} \]

All the experiments were exercised by following to the best practiced described in [19].

Comparison Works: As comparison, we select three of the most modern and promising DGF in the literature: TortoiseFuzz [5], ParmeSan [7], and SAVIOR [8]. As a baseline, we choose AFL++ [2] and AFL [1] as two of the most generic and coverage-based greybox fuzzer used in the community. Moreover, we deploy FishFuzz over QSYM [20] to answer to RQ6. For our evaluation, we choose ASan and UBSan as sanitizers as mentioned in Section V.

Experiment Setup: All the experiments where performed on a Xeon Gold 5218 CPU (22M Cache, 2.30 GHz) equipped with 64GB of memory. We evaluate all ASan targets on Ubuntu 22.04, but fall back to Ubuntu 16.04 for the SAVIOR/QSYM + UBSan evaluation due to compatibility issues of SAVIOR and QSYM with newer versions of Ubuntu. All experiments were run in docker containers with one core assigned.

Benchmarks Selected: We choose three benchmarks. Specifically, two sets of programs come from TortoiseFuzz [5] and SAVIOR [8] that we deploy over Ubuntu 22.04 and 16.04, respectively. Since we use the TortoiseFuzz benchmark set with the ASan sanitizer, we call it the ASan benchmark. Likewise, the SAVIOR benchmark contains only UBSan sanitizers, thus we name it the UBSan benchmark. Regarding the ASan benchmark, we select 9 out of 10 programs and discard two of them due to incompatibility with Ubuntu 22.04. Additionally, only for ParmeSan, we remove 3 programs from the ASan benchmark due to an non-resolvable exception in ParmeSan—while the other fuzzers handle all programs. Regarding the UBSan benchmark, we choose 7 and remove one program because it does not compile on Ubuntu 16.04. Finally, we

\( dsf(s_1, f_e) = dff(fd, fe) \)
\( dsf(s_2, fg) = 0 \)

\( dff(fd, fe) \)
\( fg \)
\( s_2' \) execution path

Fig. 4: Example of indirect call handled by FishFuzz. The picture shows a CF where \( f_c \) and \( f_d \), as well as \( f_b \) and \( g_b \), are connected through an unresolved indirect call. The execution path of the seed \( s_1 \) (red colored) traverses the indirect call between \( F_c \) and \( F_d \). This allows FishFuzz to realize the existence of a path to \( F_d \), and consequently to calculate the distance \( dsf(s_1, F_e) = dff(F_d, F_e) \), which has been already extracted at compilation time. As a special case, a function could be isolate from the CG (e.g., \( f_g \)). In this case, when a seed \( s_2 \) (green colored) this the function, we consider the distance as zero.

\( 1 \) We also successfully test it on LLVM 10.0.1

2LIBMING and CATDOC fail to compile on Ubuntu 22.04 in their latest version.

3OBJDUMP runs out of memory during compilation.
compose a benchmark of 28 real programs for the experiments in Section VI-D.

A. RQ1: How many targets does FishFuzz reach?

We want to evaluate if the exploration phases of FishFuzz can reach more targets respect similar DGFs. To this end, we set two experiments, first, we exercise ParmeSan and TortoiseFuzz against the ASan benchmark, then, SAVIOR against the UBSan benchmark. Finally, we evaluate FishFuzz against both ASan and UBSan benchmarks. For ASan, we run 5 rounds 60 hours each, while for UBSan, we run 5 rounds 24 hours each.

The results for ASan and UBSan benchmarks are in Table I and Table II, respectively. Likewise, we show the respective p-values of the Mann-Whitney U test in Table III and Table IV.

The figures show FishFuzz reaches up to 58.53% and 132.02% more edges compared with TortoiseFuzz and ParmeSan on average, respectively. Overall, FishFuzz performs better than pure coverage-guided fuzzers like AFL++ by reaching up to 47.95% more edges. The only exceptional case was EXIV2, for which FishFuzz reached −4.9% edges than AFL++ on average. This behavior can be explained due to an over-optimization of the AFL++ instrumentation that allows to exercise more seeds. Regardless a slightly drop in the exploration, FishFuzz manages to find more unique bugs respect to AFL++ (more info in Section VI-C). Furthermore, FishFuzz shows an higher coverage in the UBSan benchmark, where we reach up to 21.33% more edges compared with SAVIOR.

FishFuzz expresses better performances also in terms of reached targets. Specifically, we reach up to 38.15% and 116.63% more targets than TortoiseFuzz and ParmeSan, respectively. Similarly for the coverage. FishFuzz performs comparably with AFL++ by reaching up to 40.23% more targets. Again, here we notice a drop of −3.88% when compared with AFL++, this is expected since targets and coverage are correlated measures. Finally, we observe an improvement in the reached targets when comparing FishFuzz against SAVIOR of 19.61% on average at best.

Takeaway: Our experiment shows the exploration phase of FishFuzz can reach more targets respect to modern DGFs and with similar, if not better, results of AFL++.

B. RQ2: Does FishFuzz balance the energy?

We assess the ability of FishFuzz to redistribute energy among the targets. For this evaluation, we choose AFL as baseline because it is the base code for our prototype, therefore, we can better appreciate the improvements from our methodology. For the experiment, we run FishFuzz and AFL against the UBSan benchmark for 3 rounds of 24 hours each. Then, we average the target visit frequency. Finally, we order the targets by visit frequency and plot them in Figure 5.

The combination of the FishFuzz culling algorithm (Section III-B) and the target distance (Section IV-C) tend to re-assign energy to the lesser tested targets. This is reflected in Figure 5 where FishFuzz shows fewer targets with zero frequency in the distribution tails. Specifically, this is evident for 5 out of 7 programs in which FishFuzz expresses a better balanced energy respect to AFL (i.e., all targets have been visited at least once). For TCPDUMP, we observe FishFuzz has a few non-visited targets, but overall the curve is better redistributed. JASPER is the only case where AFL seems to have a slightly better balancing. We further investigate JASPER and notice this behavior is caused by FishFuzz that discovers new targets in the last part of the campaign. Consequently, FishFuzz has not time to assign energy to them.

Takeaway: Overall, we show that FishFuzz can effectively redistribute the energy compared with a fair baseline. This leads to a better target exploitation phase and to statistically increase the chance to find new bugs.

C. RQ3: How efficiently does FishFuzz find bugs?

In this experiment, we want to evaluate the ability of the exploitation phase in triggering bugs. To this end, we choose both ASan and UBSan benchmarks. We note FishFuzz is the only DGF that has been successfully deployed and tested against two out-of-the-shell sanitizers, while previous works considered either ASan or UBSan. Similar for Section VI-A, we exercise 5 rounds of 60 hours each in case of ASan, while 24 hours per run in case of UBSan (as in the original paper). Then, we measure the number of unique bugs for ASan and the number of triggered targets in UBSan. For ASan, we report the number of unique bugs because each bug can be associated to multiple targets, thus it could be ambiguous simply referring to the targets. Conversely, UBSan targets might not be associate to a bug, thus we prefer to indicate the targets themselves. For instance, an integer overflow in JASPER was considered as an intended behavior by the authors, and thus not considered as a bug.

For ASan, we identify unique bugs by first hashing stack traces to disambiguate crashes, followed by manually triaging bugs. For UBSan, we extract the output pattern and identify its source location (as done by SAVIOR’s authors after contacting them).

Bugs Found: The results for ASan and UBSan benchmarks are shown in Table V and Table VI respectively. For both ASan and UBSan, FishFuzz triggers more bugs/targets respect to the state-of-the-art. Specifically, FishFuzz finds 40 unique bugs in the ASan benchmark for its best round, which doubles the best rounds of AFL++ (20), TortoiseFuzz (20), and ParmeSan (9). We observe similar results also in terms of average unique bugs, where FishFuzz finds 30.6 bugs against AFL++ (16), TortoiseFuzz (16.2), and ParmeSan (9). Interestingly, even though FishFuzz reaches a lower coverage than AFL++ for EXIV2 (−4.9% – Section VI-A), the exploitation phase can focus the energy and reveal more bugs. For what concern the UBSan benchmark, we observe FishFuzz triggers more targets compared with SAVIOR. In particular, we activate from 8.21% to 62.50% more targets than SAVIOR in the best round, while from 2.99% to 52.17% more on average.

4We are planning to port FishFuzz over AFL++ for a better comparison.
Takeaway: We further measure the Time-to-Exposure in Table VII. Similar for the bug report, we only consider the time to exposure for the ASan benchmark since it manages to find new bugs, of which were confirmed CVEs. FishFuzz finds most of the bugs/CVEs in less than a day while previous works tested the same program for the equivalent of 8 weeks.

**Takeaway:** FishFuzz shows to be effective in finding new CVEs since it manages to find 18 new ones in less than week over programs already deeply tested by previous works.

**E. RQ5: How does FishFuzz redistribute exploration and exploitation?**

In this experiment, we investigate the respective contributions of the exploration (both inter- and intra-function) and exploitation phases. To this end, we run FishFuzz against the UBSan benchmark for 24 hours. Then we measure coverage, triggered targets, and trace the time evolution of the FishFuzz phases (i.e., exploration or exploitation).

In Figure 8, we show covered edges correlated with fuzzer phases. Specifically, we assign a blue background to the inter-function exploration phase, a green background to the intra-function exploration phase, and a red background to the exploitation phase. We observe two patterns. The first pattern regards coverage-growth and FishFuzz finds most of the bugs/CVEs in less than three days, while only 2 required almost seven days. We found 11 bugs with the ASan sanitizers and 3 bugs with UBSan. Specifically, most of the bugs were heap-overflow (8). We also found some assert violation (3), divide-by-zero (1), stack-exhausted (1), and shift exponential (1). Interestingly, the CVE-2022-27941 was found in less than a day while previous works tested the same program for the equivalent of 8 weeks.

**Table VIII** shows the result of our experiment. In total, we found 25 new bugs, 18 of which were confirmed CVEs. FishFuzz finds most of the bugs/CVEs in less than three days, while only 2 required almost seven days. We found 11 bugs with the ASan sanitizers and 3 bugs with UBSan. Specifically, most of the bugs were heap-overflow (8). We also found some assert violation (3), divide-by-zero (1), stack-exhausted (1), and shift exponential (1). Interestingly, the CVE-2022-27941 was found in less than a day while previous works tested the same program for the equivalent of 8 weeks.

**Takeaway:** FishFuzz shows to be effective in finding new CVEs since it manages to find 18 new ones in less than week over programs already deeply tested by previous works.

**D. RQ4: Can FishFuzz find new bugs?**

We challenge the ability of FishFuzz to find new CVEs in real applications. For this experiment, we choose 28 programs from top tiers publications, i.e., TortoiseFuzz [5], SAVIOR [8], GREYONE [22], FuzzGen [23], as well as from the fuzzing community. For each program, we deployed ASan and UB-San, respectively. We run a session one week long for each program.

Our results show that, out of 8 programs, FishFuzz finds known bugs faster. We discuss them in a dedicated section in Section VI-D. The paper claims 10 rounds of 140 hours each, which is around 8 weeks.
TABLE III: p-values of the Mann-Whitney U test from the experiments in Table I while the column UNIQ refers to the unique bugs found.

| Program | cov | AFL++ reach | uniq | | cov | TortoiseFuzz reach | uniq | | cov | ParmeSan reach | uniq |
|---------|-----|-------------|------| |     |                  |      | |     |                  |      |
| exiv2   | 0.2222 | 0.1508 | 0.4884 | 0.0119 | 0.0119 | 0.1251 | |     |                  |      |
| flvmeta | 0.0200 | 1.0000 | 1.0000 | 0.0056 | 1.0000 | 1.0000 | |     |                  |      |
| gpac    | 0.0079 | 0.0079 | 0.0114 | 0.0079 | 0.0079 | 0.0114 | |     |                  |      |
| liblouis| 0.0317 | 0.0159 | 0.0040 | 0.0159 | 0.0159 | 0.0040 | |     |                  |      |
| lbtiff  | 0.0079 | 0.0079 | 1.0000 | 0.0079 | 0.0079 | 1.0000 | |     |                  |      |
| nasm    | 0.0079 | 0.0119 | 0.1770 | 0.0079 | 0.0119 | 0.1770 | |     |                  |      |
| ngiflib | 0.0926 | 0.0937 | 1.0000 | 1.0000 | 0.2903 | 1.0000 | |     |                  |      |
| tcppreplay | 0.4005 | 0.3808 | 0.4065 | 0.0119 | 0.0937 | 0.0114 | |     |                  |      |

TABLE IV: p-values of the Mann-Whitney Y Test from the experiments in Table II while the column UNIQ refers to the targets triggered.

| Program | cov | SAVIOR reach | uniq |
|---------|-----|-------------|------|
| djpeg   | 0.1508 | 0.2222 | 0.1798 |
| jasper  | 0.0317 | 0.0079 | 0.0119 |
| readelf | 0.4019 | 0.4019 | 0.0111 |
| tcpreplay | 0.0079 | 0.0079 | 0.0119 |
| tiff2pdf | 0.0556 | 0.0556 | 0.0079 |
| tiff2ps | 0.0158 | 0.0556 | 0.0134 |
| xmllint | 0.0317 | 0.0749 | 0.0253 |

TABLE V: Unique bugs found in the ASan benchmark after 5 rounds of 60 hour each. We report the best and the average round results.

| Program | FishFuzz best | AFL++ best | TortoiseFuzz best | ParmeSan best |
|---------|---------------|------------|------------------|---------------|
| exiv2   | 5             | 3.8        | 4                | -             |
| flvmeta | 0.00.0        | 0.0        | 0.0              | 0.0           |
| gpac    | 22            | 16.0       | 7                | 6.0           |
| liblouis| 3             | 3.0        | 0                | -             |
| lbtiff  | 1             | 0.2        | 1                | 1.0           |
| nasm    | 1             | 0.4        | 0                | -             |
| ngiflib | 6             | 6.0        | 6                | 3             |
| tcppreplay | 2             | 1.2        | 2                | 1.0           |

TABLE VI: Targets triggered in the UBSan benchmark upon 5 rounds of 24 hours each. We report the best and the average round results.

Moreover, we run the experiments against the UBSan benchmark for 3 rounds of 24 hours each.

Combining QSYM+FishFuzz improves every aspect of the original fuzzer (Table IX). For instance, we improve the trigger targets up to 64.98% respect to QSYM+AFL and up to 116.07% and 41.26% for targets reached and coverage, respectively. Finally, FishFuzz also improves the number of seeds in the queue by reaching 91.54% more seeds at maximum (i.e., path column).

**Takeaway:** This experiment demonstrates that FishFuzz is compositional and helps other fuzzers improve their performance.

VII. DISCUSSION

Conversely, FishFuzz switches to exploitation when it cannot reach new targets. We can further infer this conclusion from the second pattern, where the two exploration phases occur more often at the beginning of the fuzzing campaign, while the exploitation is favored towards the end. This again represents the design of our culling algorithm (Section III-B): when the fuzzer reaches a coverage-wall [23] (e.g., a plateau), FishFuzz prefers the exploitation phase to find more bugs in already reached targets.

**F. RQ6: Can other fuzzers benefit from our strategies?**

To answer the question if other fuzzer can profit from our strategies, we combine QSYM [20] with FishFuzz and AFL to measure if our cull queuing improves the performance. Specifically, we run the QSYM with one AFL-primary and one concolic executor (more details in the original paper [20]).

Moreover, we run the experiments against the UBSan benchmark for 3 rounds of 24 hours each.

Combining QSYM+FishFuzz improves every aspect of the original fuzzer (Table IX). For instance, we improve the trigger targets up to 64.98% respect to QSYM+AFL and up to 116.07% and 41.26% for targets reached and coverage, respectively. Finally, FishFuzz also improves the number of seeds in the queue by reaching 91.54% more seeds at maximum (i.e., path column).

**Takeaway:** This experiment demonstrates that FishFuzz is compositional and helps other fuzzers improve their performance.

Here, we discuss limitations and future work for FishFuzz. Specifically, we focus on performance (Section VII-A), target size (Section VII-B), and combination with orthogonal techniques (Section VII-C).

A. Performance

Our prototype extends AFL with instrumentation to trace functions and targets information. During our extensive experiments, we notice our prototype does not achieve optimal performances in some cases (EXIV2 in Table I), adversely affecting the fuzzing campaign. We tracked the root cause to two problems: First, AFL++ uses more efficient instrumentation compared to AFL. Second, FishFuzz uses additional shared memory to trace explored functions and reached targets. Synchronizing with this structure introduces latency and reduces the number of seeds exercised. To overcome these limitations, we are currently porting our prototype to AFL++
to leverage the more efficient instrumentation. We realized this limitation during our evaluation against AFL++. With this (engineering) optimization, FishFuzz performance will further improve and we will integrate those results.

B. Target Set Size

FishFuzz is designed to handle large target sets (up to 20k in our experiments [Section VI]). Even though FishFuzz scales up efficiently in these cases, we also observe a drop of performances for small target sets (e.g., at around tens targets). We plan to tackle this problem in two directions. First, we could employ different mutators to direct seeds faster, such as [24]. Second, we believe this observation suggests the need of specific ad-hoc seed-distance metrics according to the context. Therefore, we will investigate the performance of different seed-target metrics and infer the best trade-off as future work.

C. Combining FishFuzz with other works

BEACON [25] (to be published at Oakland’22), a concurrent DGF uses software analysis (e.g., static or dynamic) to foresee (and discard) unreachable portions of code. FishFuzz would benefit from these techniques to speed up the initial exploration phase or better re-assign energy to targets. Similarly, we consider to combine SAVIOR [8] with our exploration phase to investigate if different approaches can lead to better performances.

VIII. RELATED WORKS

FishFuzz improves existing fuzzing work across two research areas: Directed Greybox Fuzzers (Section VIII-A) and Multistage Fuzzers (Section VIII-B).

A. Directed Greybox Fuzzers

DGF is a branch of fuzzing that specializes fuzzers for hitting a given set of targets (instead of improving code-coverage).

Böhme et al. discusses the first prototype, AFLGo [11], which models the distance seed-targets as an harmonic average distance. However, the AFLGo approach losses precision for large target sets. In this regard, FishFuzz relies on a novel seed-target distance whose precision is not affected by the number of targets. Improvements to AFLGo were further proposed by Chen et al. with Hawkeyes [7] and Peiyuan et al. with FuzzGuard [26]. These works try to handle indirect calls by adopting heavy weight static analysis (Hawkeyes) or using deep learning to discard unfruitful inputs (FuzzGuard), respectively. Conversely, FishFuzz does not need any complex
TABLE VIII: Results of one week of fuzzing over 28 Real-World application. We discover 18 confirmed CVEs, while two are pending, and two are classified as only-bug.

| Project | Program/Driver | Type | Sanitizer | Ref | CVE status | time | benchmark |
|---------|----------------|------|-----------|-----|------------|------|-----------|
| libca   | img2txt        | divide by zero | ASan | issue_65 | CVE-2022-0856 | <1d | GREYONE   |
| libca   | lou_checktable  | heap-overflow | ASan | issue_1171 | CVE-2022-26981 | <3d | GREYONE   |
| libca   | lou_trace       | out-of-bound read | ASan | issue_1214 | CVE-2022-31783 | <3d | GREYONE   |
| tcpreplay| tcprewrite      | heap-overflow | ASan | issue_718  | CVE-2022-27940 | <1d | TortoiseFuzz |
| tcpreplay| tcprewrite      | assert    | ASan | issue_717  | CVE-2022-27939 | <1d | TortoiseFuzz |
| tcpreplay| tcprewrite      | heap-overflow | ASan | issue_716  | CVE-2022-27941 | <1d | TortoiseFuzz |
| tcpreplay| tcprewrite      | heap-overflow | ASan | issue_719  | CVE-2022-27942 | <3d | TortoiseFuzz |
| gmp     | MP4Box         | heap-overflow | ASan | issue_2138 | CVE-2022-26967 | <3d | TortoiseFuzz |
| gmp     | MP4Box         | heap-overflow | ASan | issue_2173 | CVE-2022-29517 | <3d | TortoiseFuzz |
| libmpeg2| mpeg2_dec_fuzzer | memcpy overlap | ASan | issue_2179 | CVE-2022-30976 | <3d | TortoiseFuzz |
| mujs    | mujs-pp        | null pointer dereference | ASan | issue_161-1 | bug-only | <1d | EMS       |
| mujs    | mujs-pp        | null pointer dereference | ASan | issue_161-2 | CVE-2022-30975 | <1d | EMS       |
| mujs    | mujs           | stack exhausted | ASan | issue_162  | CVE-2022-30974 | <3d | EMS       |
| sox     | sox            | reachable assertion | ASan | issue_360-1 | CVE-2022-31650 | <1d | MoonLight |
| libavc  | avc_enc_fuzzer | assert    | UBSan | issue_223984040 pending | <1d | FuzzGen   |
| libavc  | avc_enc_fuzzer | heap-overflow | UBSan | issue_224160472 pending | <3d | FuzzGen   |
| Bento4  | mp4tag         | heap-overflow | ASan | issue_677  | CVE-2022-27607 | <3d | others    |
| Bento4  | mp42hevc       | heap-overflow | ASan | issue_678  | CVE-2022-27607 | <3d | others    |
| libsvgx | img2svgx       | assert    | ASan | issue_163  | CVE-2022-27938 | <1d | GREYONE   |
| binutils| nm-new         | stack exhausted | ASan | issue_28995 | CVE-2022-27943 | <7d | SAVIOR    |
| jasper  | jasper         | shift exponent exceed | ASan | issue_311  | CVE-2022-29458 | <1d | GREYONE   |
| ncurse  | tic            | out-of-bound read | ASan | mail list  | CVE-2022-29458 | <1d | GREYONE   |
| ncurse  | tic            | heap-overflow | ASan | mail list  | CVE-2022-29458 | <1d | GREYONE   |

TABLE IX: Running QSYM+FishFuzz against QSYM+AFL in the UBSan benchmark. The results refer to the average of 3 rounds for 24 hours each. Column path represents the number of seeds in the queue.

| Program | path | QSYM+AFL | path | QSYM+FishFuzz | path | vs QSYM+AFL |
|---------|------|----------|------|--------------|------|-------------|
| jpeg    | 1304 | 10303    | 3310 | 85.7         | 2497 | 12029       |
| jpeg    | 1090 | 9730     | 1451 | 29.3         | 1778 | 11005       |
| jpeg    | 542  | 2580     | 232  | 22.0         | 666  | 2541        |
| jpeg    | 1811 | 9941     | 1068 | 62.7         | 2323 | 10048       |
| jpeg    | 1741 | 12759    | 1950 | 71.1         | 2652 | 18022       |
| jpeg    | 1188 | 8832     | 960  | 11.0         | 1839 | 10228       |
| jpeg    | 813  | 6181     | 496  | 7.3          | 1504 | 8365        |
| jpeg    | 2232 | 8103     | 672  | 9.3          | 2741 | 8561        |

analysis to resolve indirect jumps, while its seed selection automatically promotes interesting inputs.

Steps toward more scalable DGF are discussed by Österlund with ParmeSan [3] and Chen with SAVIOR [8], respectively. Both ParmeSan and SAVIOR consider as targets all the sanitizers labels. Additionally, SAVIOR introduces an heavy reachable analysis to select interesting inputs. Both works suffer from the original AFLGo limitation since they collapse the distance seed-targets into a scalar, thus losing precision. FishFuzz differs from these works for two reasons: first, it employs a novel distance seed-targets that overcomes scalability limitations, second, it uses a faster exploration phase to boost the targets discovery.

Gwangmu et al. propose CAFL [27] (Constraint guided directed greybox fuzzing). The goal of this work is to synthesis a POC from a given crash by following a similar approach of AFLGo. In their scenario, CAFL considers only one target, while FishFuzz is designed to handle a large number of targets. Finally, Xiaogang et al. discuss Regression Greybox Fuzzing [12], their work contains methods to select possible bogus code locations by analyzing the repository history. This approach is then combined with a more efficient power schedule policy. We consider this work as orthogonal to FishFuzz since we focus on the seed selection strategy, while they recognize interesting code locations for testing.

Huang et al. introduces BEACON [25], which uses sophisticated static-analysis to remove unfeasible paths, thus speeding up the exploration phase. Conversely, FishFuzz aims at improving the exploitation phase and trigger targets. We consider their approach as orthogonal to FishFuzz, we further plan to combine the two strategies in the future.

B. Multistage Fuzzers

Multistage fuzzers use exploration and exploitation phases to reach and trigger multiple targets. Böhme et al. proposes AFLFast [28], which relies on Markov chain to probabilisti-
FishFuzz re-assigns energy to lesser explored targets, thus resulting in a tail with lesser non-tested targets. For JASPER, we observe a few targets with zero visits because discovered in the latest phase of the campaign.

Lemieux et al. [29] study new mutation strategies and seed selections to hit rare branches. Their contribution is mainly energy distribution related, while queue culling and seed distance are not discussed.

Wang et al. [5] select interesting targets upon extensive software analysis, that are then combined with a novel queue culling strategy. However, their approach considers only time-invariant targets, thus not adapting the fuzzer energy toward more promising code locations. Conversely, the queue culling mechanism of FishFuzz is adaptive and can be potentially used to improve the performances of Wang’s work.

IX. CONCLUSION

Directed Greybox Fuzzing has been hampered by averaged distance metrics that over-eagerly aggregate paths into scalars and simple energy distribution that simply assigns equal energy to all targets in a round robin fashion.

We draw inspiration from trawl fishing where a wide net is cast and pulled to reach many targets before they are harvested. FishFuzz improves the exploration and exploitation phases with explicit feedback for both phases and a dynamic switching strategy that alternates mutation and energy distribution based on the current phase. Additionally, our dynamic target ranking automatically discards exhausted targets and our novel multi-distance metric keeps track of tens of thousands of targets without loss of precision.

We evaluate FishFuzz against 44 programs and have, so far, discovered 25 new bugs (18 CVEs). FishFuzz will be released as open source and we provide a test environment to play with our novel DGF.

REFERENCES

[1] Z. Michal, “american fuzzy lop,” https://lcamtuf.coredump.cx/afl/, 2013.
[2] 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). USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi
[3] S. Osterlund, K. Razavi, H. Bos, and C. Giuffrida, “{ParmeSan}: Sanitizer-guided greybox fuzzing,” in 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 2289–2306.
Fig. 8: FishFuzz stages in 24 hour of fuzzing.  for exploitation.  for inter-function exploration, and  for intra-function exploration.

[4] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder, “High system-code security with low overload,” in 2015 IEEE Symposium on Security and Privacy. IEEE, 2015, pp. 866–879.

[5] 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.

[6] 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.

[7] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, “Hawkeye: Towards a desired directed grey-box fuzzer,” in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018, pp. 2095–2108.

[8] 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.

[9] D. Babic, S. Bucur, Y. Chen, F. Ivancic, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, “Fudge: Fuzz driver generation at scale,” in Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019.

[10] P. Godefroid, M. Y. Levin, and D. Molnar, “Sage: Whitebox fuzzing for security testing: Sage has had a remarkable impact at microsoft.” Queue, vol. 10, no. 1, pp. 20–27, 2012.

[11] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, “Directed greybox fuzzing,” in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 3239–3244.

[12] X. Zha and M. Böhme, “Regression greybox fuzzing,” in Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, 2021, pp. 2169–2182.

[13] 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), 2019, pp. 1–15.

[14] Google, “Addresssanitizer,” https://github.com/google/sanitizers/wiki/AddressSanitizer 2014.

[15] K. Serebryany, D. Bruneau, A. Potapenko, and D. Vyukov, “([AddressSanitizer]: A fast address sanity checker,” in 2012 USENIX Annual Technical Conference (USENIX ATC 12), 2012, pp. 309–318.

[16] Google, “Undefinedbehaviorsanitizer,” https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html 2017.

[17] llvm, “The LLVM Compiler Infrastructure Project,” http://llvm.org/.

[18] D. B. Johnson, “A note on dijkstra’s shortest path algorithm,” Journal of the ACM, vol. 10, no. 1, pp. 20–27, 2012.

[19] E. van der Kouwe, G. Heiser, D. Andriesse, H. Bos, and C. Giuffrida, “SoK: Benchmarking Flaws in Systems Security,” in EuroS&P, Jun. 2019. [Online]. Available: Paper=https://download.vusec.net/papers/benchmarking-crimes Eurosp19.pdfSlides=https://www.vusec.net/wp-content/uploads/2019/06/Benchmarking-Flaws-in-Systems-Security-EuroSP2019.pdfWeb=https://www.vusec.net/projects/benchmarking-crimesPress=https://bsd.ly/3knXxKk

[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). Baltimore, MD: USENIX Association, Aug. 2018, pp. 745–761. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[21] P. Bardou, J. Marieote, F. Escudier, C. Djemiel, and C. Klop, “jvenn: an interactive venn diagram viewer,” BMC bioinformatics, vol. 15, no. 1, pp. 1–7, 2014.

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

[23] K. Isopoglou, D. Austin, V. Mohan, and M. Payer, “[FuzzGen]: Automatic fuzzer generation,” in 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 2271–2287.

[24] 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.

[25] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, “Beacon: Directed greybox fuzzing with provable path pruning.” in 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 2577–2594.

[26] P. Bardou, J. Marieote, F. Escudier, C. Djemiel, and C. Klop, “jvenn: an interactive venn diagram viewer,” BMC bioinformatics, vol. 15, no. 1, pp. 1–7, 2014.

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

[28] K. Isopoglou, D. Austin, V. Mohan, and M. Payer, “[FuzzGen]: Automatic fuzzer generation,” in 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 2271–2287.

[29] 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.

[30] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, “Beacon: Directed greybox fuzzing with provable path pruning.” in 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 2577–2594.

[31] P. Bardou, J. Marieote, F. Escudier, C. Djemiel, and C. Klop, “jvenn: an interactive venn diagram viewer,” BMC bioinformatics, vol. 15, no. 1, pp. 1–7, 2014.

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

[33] K. Isopoglou, D. Austin, V. Mohan, and M. Payer, “[FuzzGen]: Automatic fuzzer generation,” in 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 2271–2287.

[34] 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.

[35] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, “Beacon: Directed greybox fuzzing with provable path pruning.” in 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 2577–2594.

[36] P. Bardou, J. Marieote, F. Escudier, C. Djemiel, and C. Klop, “jvenn: an interactive venn diagram viewer,” BMC bioinformatics, vol. 15, no. 1, pp. 1–7, 2014.

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

[38] K. Isopoglou, D. Austin, V. Mohan, and M. Payer, “[FuzzGen]: Automatic fuzzer generation,” in 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 2271–2287.

[39] 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.

[40] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, “Beacon: Directed greybox fuzzing with provable path pruning.” in 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 2577–2594.