FISHFUZZ: Catch Deeper Bugs by Throwing Larger Nets

Han Zheng[1,2,6], Jiayuan Zhang[1,3], Yuhang Huang[1], Zezhong Ren[1,6], He Wang[4], Chunjie Cao[5], Yuqing Zhang[1,6,5,4], Flavio Toffalini[2] and Mathias Payer[2]

[1]*National Computer Network Intrusion Protection Center, University of Chinese Academy of Science*
[2]*School of Computer and Communication Sciences, EPFL*
[3]*School of Computer and Communication, Lanzhou University of Technology*
[4]*School of Cyber Engineering, Xidian University*
[5]*School of Cyberspace Security, Hainan University*
[6]*Zhongguancun Laboratory*

Abstract

Fuzzers effectively explore programs to discover bugs. Greybox fuzzers mutate seed inputs and observe their execution. Whenever a seed reaches new behavior (*e.g.,* new code or higher execution frequency), it is stored for further mutation. Greybox fuzzers directly measure *exploration* and, by repeating execution of the same targets with large amounts of mutated seeds, passively *exploit* any lingering bugs. Directed greybox fuzzers (DGFs) narrow the search to a few code locations but so far generalize distance to all targets into a single score and do not prioritize targets dynamically.

FISHFUZZ introduces an input prioritization strategy that builds on three concepts: (i) a novel multi-distance metric whose precision is independent of the number of targets, (ii) a dynamic target ranking to automatically discard exhausted targets, and (iii) a smart queue culling algorithm, based on hyperparameters, that alternates between *exploration* and *exploitation*. FISHFUZZ enables fuzzers to seamlessly scale among thousands of targets and prioritize seeds toward interesting locations, thus achieving more comprehensive program testing. To demonstrate generality, we implement FISHFUZZ over two well-established greybox fuzzers (AFL and AFL++). We evaluate FISHFUZZ by leveraging *all* sanitizer labels as targets. In comparison to modern DGFs and state-of-the-art coverage guided fuzzers, FISHFUZZ reaches higher coverage compared to the direct competitors, finds up to 2.8x more bugs compared with the baseline and reproduces 68.3% existing bugs faster. FISHFUZZ also discovers 56 new bugs (38 CVEs) in 47 programs.

1 Introduction

Greybox fuzzing is an established automated program testing technique aimed at finding bugs. Fuzzing 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 reproduction of the crash later during debugging. This effectiveness resulted in considerable interest in more sophisticated designs [2, 5–8, 10, 21, 25, 27, 31].

Successful greybox fuzzers balance two aspects: *exploration* and *exploitation*. During *exploration*, the main goal is to increase coverage, *i.e.,* creating new seeds that reach new program areas [8, 31]. During *exploitation*, the main goal is to trigger bugs by executing a piece of code with diverse inputs, *i.e.,* mutating existing seeds to trigger bugs in these already covered program areas. Unfortunately, fuzzers have no direct feedback for *exploitation* and assume that random mutations will execute the same code *sufficiently well*. Directed Greybox Fuzzers (DGF [3]) prioritize exploitation by directing exploration towards a specific code location (target), improving the likelihood to find bugs at that location [3, 5, 7, 21]. DGFs leverage the distance between seeds and targets to prioritize inputs more likely to trigger errors.

Recent DGFs try to balance *exploitation* and *exploration* to automatically trigger bugs in larger sets of targets (on the order of thousands) [5, 7, 21, 32]. However, we observe two core limitations. First, they model the distance between seeds and targets as a single harmonic average. This means that, regardless of the number of targets, they always collapse the seed-target distance into a single scalar. Therefore, for large target sets, the distance results are distorted, and the fuzzer loses precision. DGFs average the distance because tracing each target individually (*e.g.,* at the basic block level) results in prohibitive overhead. Second, current works assign 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 the priority of a target changes during the fuzzing campaign and a fuzzer must adjust its strategy as some targets may be easier to reach than others. 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 miss-classify the importance of a code location and waste computing resources.

Observing these challenges, we propose FISHFUZZ that (i) estimates a more precise distance between seeds and targets, (ii) dynamically ranks interesting targets among thousands,

and (iii) automatically prioritizes inputs. FISHFUZZ builds on three key contributions: (1) a novel distance metric more robust to indirect jumps that can select the "closest" seed for each target (thus improving exploration), (2) a dynamic target ranking that automatically discards exhausted targets and steers the fuzzer's energy towards (more) promising locations (thus improving exploitation), and (3) a novel queue culling algorithm to efficiently orchestrate between *exploration* and *exploitation*. 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 each target is explored. Unlike previous works, FISHFUZZ easily handles tens of thousands of targets without loss of precision (*e.g.,* in our largest target, we cover over 20*k* *targets*). Concurrently, FISHFUZZ automatically prioritizes rarely tested targets, which are more likely to contain bugs [18, 25]. The combination of these three 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 improved bug-finding capabilities.

We show the generality of FISHFUZZ by deploying our strategies over two well-established greybox fuzzers: AFL and AFL++. Then, we compare our prototypes against three modern DGFs (*i.e.,* ParmeSan [21], TortoiseFuzz [27], and SAVIOR [7]), four modern two-stage fuzzers (*i.e.,* AFLFast [4], FairFuzz [17], EcoFuzz [29], K-Scheduler [23]) as well as against AFL++ [8] and AFL [31]. We conduct experiments over *four* benchmark sets. The first two are composed of 11 and 8 programs taken from TortoiseFuzz [27] and SAVIOR [7], respectively. For the third one, we choose 7 programs that cover different input formats, *e.g.,* document, image, text, executable. The last benchmark selects 30 real-world programs from other top-tier fuzzing works. During *exploration*, we achieve higher coverage (up to 146% more) and hit more targets (up to 112% more) compared with the state-of-the-art. For *exploitation*, FISHFUZZ triggers 2.8x more targets and finds up to 2x unique bugs in comparison to its competitors. More precisely, FISHFUZZ reproduces 99 unique previous bugs, among which 71 (68.3%) faster than previous works. Moreover, we discover 56 new bugs from which 38 are already confirmed as CVEs. Finally, we investigate the different bugs discovered when deploying FISHFUZZ over AFL or AFL++. Our results show that the base fuzzer influences the type of bugs discovered (*i.e.,* AFL/FISHFUZZ finds different bugs than AFL++/FISHFUZZ). This insight suggests FISHFUZZ enhances the existing bug discovery capabilities by redirecting their energy more efficiently.

To sum up, our contributions are:

- FISHFUZZ: a novel input prioritization strategy that maximizes the number of *explored* and *exploited* targets.
- A distance metric between seeds and targets that is independent of the size of the target set and robust against unresolved indirect jumps.
- A dynamic target ranking that automatically guides the fuzzer's energy towards promising locations, while discarding thoroughly explored ones.
- A detailed evaluation against the state-of-the-art and the discovery of 56 bugs (38 CVEs) in 47 programs.

The full source code is released at `https://github.com/HexHive/FishFuzz`. The artifact with a demonstration is available at `https://zenodo.org/record/6405418`.

2 Background

This section introduces background information of input prioritization, upon which we build FISHFUZZ (§2.1). Moreover, we define FISHFUZZ's scope (§2.2).

2.1 Input prioritization

Modern fuzzers, such as AFL [31] and AFL++ [8], take a program as input and a set of inputs (seeds) to submit to the program. Their workflow is loop-based: they select seeds, mutate them, and submit the mutated inputs 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 *favored*, to indicate whose seeds will be selected in the next fuzz iteration. The *favored* setting can follow various strategies depending on the targeted results. For instance, we can select seeds to improve the coverage, or we can guide the testing toward specific code locations in an attempt to trigger a specific bug.

AFL [31] maintains a map (`top-rated`) that pairs the visited edge and the *best* input for visiting it, where the best input is simply the smallest and fastest one to reach that edge. While executing inputs, AFL traces the visited edges and assigns the new input to the edges in top-rated if the input is smaller or can reach these edges faster. In this way, AFL can easily find a suitable input for an edge through a fast look-up. More advanced fuzzers, such as Angora [6] or AFL-Sensitive [26], include additional information, such as the calling stack, the memory access address, and the n basic blocks execution path. More recent fuzzers [27] infer the best seed based on a combination of the static analysis and the seed's execution

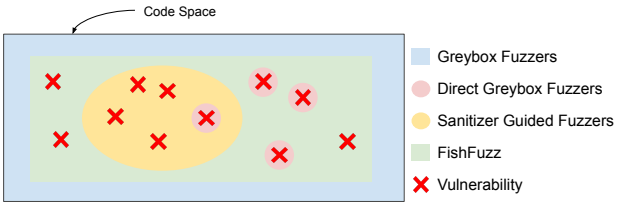


Figure 1: The image depicts the scope of FISHFUZZ relative to other fuzzer families. Greybox Fuzzers target the whole code space, Direct Greybox Fuzzers target specific targets, Sanitizer-Guided Fuzzers target the sanitizer label but not all, FISHFUZZ targets all the sanitizer labels.

Table 1: Number of targets used by ParmeSan and FISHFUZZ.

programs	ParmeSan	FISHFUZZ	Rate
flvmeta	300	4157	13.86
MP4Box	6809	107930	15.85
lou_checktable	127	1853	14.59
tiff2pdf	1015	14938	14.72
nasm	899	9160	10.19
nm-new	1813	41901	23.11
gif2tga	10	474	47.40
sum	10973	180413	16.44

path. Regardless of their approaches, we observe that current queue culling algorithms estimate the target's priority ahead-of-time, without reconsidering them during the campaign. Therefore, if a portion of code has been triggered, the cull queue keeps selecting seeds for that location, thus wasting energy. Conversely, FISHFUZZ dynamically deprioritizes the targets during the fuzzing campaign (§4.2).

2.2 FISHFUZZ's scope

Figure 1 depicts the relation of current fuzzer families with FISHFUZZ. The outer box represents the whole code space, which might contain vulnerabilities (represented as red crosses). Fuzzers try to trigger vulnerabilities by *exploring* code and *exploiting* the vulnerabilities. Greybox fuzzers focus on *exploration*: covering the entire code space regardless of the presence of vulnerabilities. As an opposite strategy, Direct Greybox Fuzzers (DGFs) [3] try to reach and trigger targets in a target set instead of the whole code space. The target set is usually manually defined. DGFs achieve this goal by modeling *exploration* and *exploitation* as distinct phases. First, they mutate seeds so that they hit the targets (*exploration* phase). Second, they try to trigger the targets (*exploitation* phase). Specifically, the *exploration* phase uses seed-target distance metrics to infer how "far" a seed is from a given target set, while *exploitation* relies on standard fuzzing techniques to "hammer" the reached locations. However, their seed-target metrics average the distance from a seed to the whole target set, thus losing precision if the target set contains more than one target. As an extension of DGF, Sanitizer-guided Fuzzers (SGF) [7, 21] stretch the target set over any location labeled by sanitizers, thus considering thousands of targets. SGFs ideally consider all labeled code as possible buggy and then direct the exploration toward those locations in an attempt to trigger an exception. However, SGFs inherit similar seed-target distance from DGFs, thus suffering from similar precision issues. Therefore, SGFs propose static analysis and heuristics to focus on promising targets to make the problem tractable. In practice, SGFs might discard relevant vulnerabilities or test code locations that are already bug-free [21]. Given this background, FISHFUZZ extends and improves upon DGFs and SGFs by handling arbitrarily large target sets without losing precision or discarding real bugs. Unlike previous works, FISHFUZZ considers all the sanitizer labels (*e.g.,* from ASan [11] and UBSan [12]) and handles 16x and 3x more targets compared with Parmesan [21] and SAVIOR [7] respectively, which are state-of-the-art SGFs (Table 1 and Table 2). To achieve our goal, we must solve the following three challenges:

Table 2: Number of targets used by SAVIOR and FISHFUZZ.

programs	SAVIOR	FISHFUZZ	Rate
djpeg	860	4927	5.73
jasper	2576	4194	1.63
readelf	236	3160	13.39
tcpdump	4114	10221	2.48
tiff2pdf	333	4235	12.72
tiff2ps	333	2740	8.23
xmllint	2627	5567	2.12
sum	11079	35044	3.16

C1: Designing a distance metric to improve the precision in the *exploration* phase. Our metric allows a fuzzer to select the closest seed for each function containing targets, thus improving precision. Conversely, current distance metrics employed in DGF and SGF average over multiple targets, thus losing precision with the growth of the target set.

C2: Designing a mechanism to dynamically select interesting targets and discard unpromising ones, thus improving the *exploitation* phase. Conversely, current SGFs use either heuristics or imprecise analysis that might (i) ignore interesting code locations and (ii) waste energy toward already triggered targets.

C3: Designing a smart queue culling algorithm that orchestrates *exploration* and *exploitation* to maximize the targets reached and triggered.

3 FISHFUZZ's Design

The design of FISHFUZZ revolves around the concepts of distance measurement and dynamic target selection, and poses the base for our queue culling algorithm in §4. Specifically, we first formalize the structures for modelling targets' priority in §3.1, then we introduce our distance measurement in §3.2.

3.1 Dynamic Target Priority

One of FISHFUZZ's key features is to focus the fuzzer's energy by selecting more promising targets. To achieve this, we associate to each target t a triple of meta-data $(hit\_frequency, reached, triggered)$, where *hit_frequency* indicates how often a target has been traversed by the seeds, *reached* if the target has been executed at least once, and *triggered* if the execution of the target crashes. The targets' metadata are organized in a set T. During the campaign, we update T through a shared memory region between the fuzzer and the tested program. Then, FISHFUZZ selects seeds to hit less-explored targets, *i.e.,* those with low *hit_frequency*. Since the selection is intertwined with the queue algorithm, we detail the process in §4.2.

3.2 Distance Measurement

To overcome the imprecision of current seed-target distance metrics [3, 21, 27], we propose to estimate the "closest" seed for each target independently. However, tracing the distance at the basic block level would require an enormous amount of information, which would slow down the fuzzing campaign. To reduce the tracing overhead, we measure the minimal distance between the seed and functions on the call graph. This approach reduces the number of required instrumentations. Finally, we rely on standard fuzzing techniques for the inner function exploration (see §4.2).

Overall, we first pre-calculate a static map that contains the minimum distance among functions, which we call *static distance map*. Then, we leverage the *static distance map* to estimate the closest seed to a function at runtime (*i.e.,* the seed that visits the function closest to the target). In the last part of the section, we discuss how to handle indirect calls.

Static Distance Map. The *static distance map* is a look-up table that contains the minimum distance for each function pair. The distance is estimated over the control-flow-graph (CFG) and the call-graph (CG), which we extract during compilation from LLVM-IR [19]. This initial analysis, for now, is oblivious to indirect calls.

To calculate the *static distance map*, FISHFUZZ 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 follows:

$$weight(f_i, f) = \begin{cases} \min dbb(m, m_f) & \text{if } \exists m_f \in f_i \\ \infty & \text{otherwise,} \end{cases} \quad (1)$$

where m is the prologue of the function f_i, and m_f is a basic block belonging to f_i and contains 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 (∞). To handle multiple function calls to f, we consider the minimum distance to leave f_i only.

Once the *weights* are computed, FISHFUZZ pre-computes the distance between two functions as the sum of their *weight* along the shortest path between two functions based on the compilation-extracted CG. Formally speaking, the distance between two functions f_a and f_b is defined as follow:

$$dff(f_a, f_b) = \sum_{f_i \in sp(f_a, f_b)} weight(f_i, f_{i+i}), \quad (2)$$

where $sp(f_a, f_b)$ is the shortest path between f_a and f_b using Dijkstra's algorithm [15], and $weight(f_i, f_{i+1})$ is Eq. 1 over two consecutive functions in the path.

Dynamic Seed to Function Distance. Having the *static distance map*, we define a function $dsf(s, f)$ that represents the distance between the functions traversed by the seed s and a function f as follows:

$$dsf(s, f) = \begin{cases} \min_{f_s \in \xi(s)} dff(f_s, f) & \text{if } f \notin \xi(s) \\ 0 & \text{otherwise,} \end{cases} \quad (3)$$

where $\xi(s)$ represents the functions traversed by the execution of the seed s. If f is traversed by the execution path of s, we consider the distance as *zero*.

With the dynamic seed distance, FISHFUZZ chooses the closest seed for a target function based on the intuition that a seed closer to a target has higher probability to reach it. We mainly use *dsf* in the inter-function exploration (§4.2).

Indirect Call handling. Unresolved indirect calls might stop a fuzzer from reaching interesting code regions. In FISHFUZZ, the *dynamic seed to function distance* (Eq. 3) already provides an approximation of indirect calls without the burden to resolve them. We illustrate this property in Figure 2, which shows a CG where the functions pair f_c - f_d, and f_b - f_g, are connected through an (unresolved) indirect call. This example contains three cases: ID1 and ID2 show when FISHFUZZ can resolve the seed-target distance, while ID3 exemplifies when FISHFUZZ cannot resolve the distance.

Case ID1. We assume the fuzzer has generated a seed s_1 that traverses f_a, f_c, f_d, and finally hits f_f (red). When

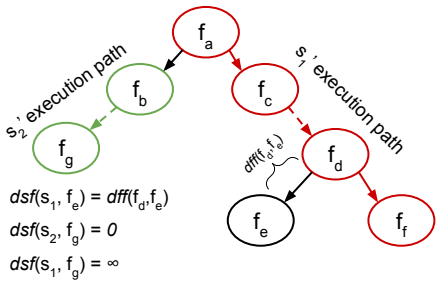


Figure 2: Example of handling indirect calls. In this CG f_c - f_d and f_b - f_g are connected through unresolved indirect calls. The execution path of the seed s_1 (red) traverses the indirect call between f_c and f_d. This allows FISHFUZZ to notice the existence of a path to f_d, and then calculate the distance $dsf(s_1, f_e) = dff(f_d, f_e)$. If a function is isolated from the CG (*e.g.,* f_g), we may exercise a seed s_2 (green) that hits the isolated function. In this case, we consider the distance $dsf(s_2, f_g) = 0$. If the seed has not resolved jumps toward a function, such as for s_1 and f_g, then we consider its distance $dsf(s_1, f_g) = \infty$.

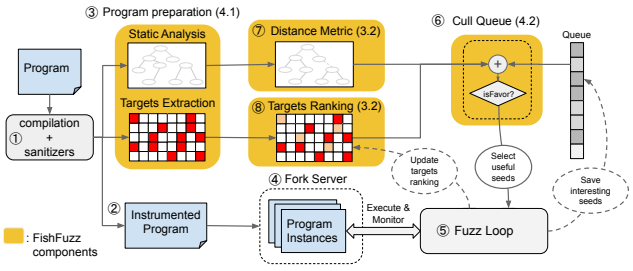


Figure 3: Overview of FISHFUZZ workflow.

this occurs, FISHFUZZ has sufficient information to compute the distance between s and f_e, which is exactly the distance between f_d and f_e (Eq. 2). Our approach is an approximation because there could be other hypothetical hidden paths behind not-yet-resolved indirect calls.

Case ID2. In this case, f_g has no direct calls to resolve the distance calculation. Therefore, we assume the fuzzer generates a seed s_2 that traverses f_g (green). According to Eq. 3, the distance is *zero* since s_2 hits the target function.

Case ID3. The execution path of s_1 does not reach any function directly connected to f_g. In this case, we set the distance between s_1 and f_g as infinite (∞), and consider f_g unreachable by s_1.

Generally, whenever a seed traverses an indirect call, FISHFUZZ can use nearby (connected) functions to estimate the minimum distance between the seed and the target function (Case ID1). If a function has no direct connections, we have two cases (i) the fuzzer generates a seed that traverses the function (Case ID2), or (ii) the function is unreachable for a given seed (Case ID3).

4 FISHFUZZ's Queue Culling Strategy

Our queue culling strategy leverages the dynamic target priority (§3.1) and the distance measurement (§3.2) to address both challenges described in §2.2. FISHFUZZ allows DGFs to scale *exploration* and *exploitation* over programs with thousands of targets without losing precision. Most importantly, we design the queue culling to be easily integrated into modern Greybox Fuzzers [31]. To prove our claims, we implement and test FISHFUZZ over AFL and AFL++. This section focuses on the design and expands crucial queue components, while we detail the implementation in §5 and evaluate in §6.

The overall workflow is depicted in Figure 3 and follows the standard fuzzing procedures [8, 21, 27, 31]. Given a target program, we compile and instrument it with specific sanitizers ①, this phase makes the instrumented program ready for the fuzzing campaign ②, and extracts initial information useful for FISHFUZZ ③. Then, 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 our queue culling ⑥ which relies on our distance measurement ⑦ and ranking ⑧.

As previously stated, the majority of the workflow follows the standard greybox fuzzers [4, 8, 31], only program preparation §4.1 and queue culling §4.2 differ. We discuss these components in the rest of the section.

4.1 Program Preparation

In the program preparation 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 [3, 6, 21, 27, 31]. Finally, we perform static analysis over LLVM-IR [19] to build the *static distance map* (§3.2). The FISHFUZZ design is agnostic to the sanitizer, in our experiment, we successfully tested ASan [11, 22] and UBSan [12]. Alternatively, one can use other targets such as assertions. We opt for sanitizer targets due to their straightforward security implications.

4.2 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 no new 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/trig-

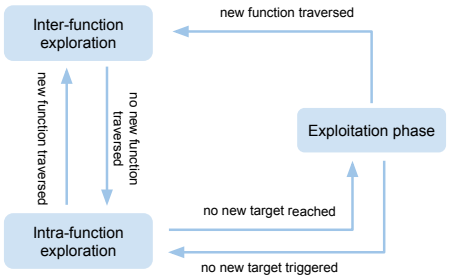


Figure 4: FISHFUZZ uses an exploration phase to select seeds closer to targets, while the exploitation phase focuses on the targets triggering. Additionally, FISHFUZZ uses two sub-exploration phases, one is specialized to reach larger number of functions (*inter-function*), while the second to maximize the reached targets (*intra-function*).

gered), we adopt a multiphase approach as suggested from previous works [21]. Specifically, FISHFUZZ relies on three phases: *inter-function exploration*, *intra-function exploration*, and *exploitation* – all pictured in Figure 4. The purpose of the *inter-function* exploration is to reach interesting functions (*i.e.,* expanding the net). This phase leverages the Distance Measurement (§3.2) to select seeds closer to functions containing untriggered targets (thus addressing **C1**). *Intra-function* exploration focuses on testing internal functions based on the original fuzzer's algorithm and tries to hit as many targets as possible (*i.e.,* start closing the net). Finally, the *exploitation phase* maximizes the number of targets triggered (*i.e.,* catching the fish). This phase uses the Dynamic Target Ranking (§3.1) to maximize the triggered targets (thus addressing **C2**).

The switch among the different phases happens at specific events: (i) every time *a new function is traversed* (*i.e., inter-function exploration*), (ii) if *no new function is traversed* for a period of time (*i.e., intra-function exploration*), (iii) if *no new target is reached* for a period of time (*i.e., exploitation*), and (iv) if *no new target is triggered* for a period of time. Swinging among the events allows FISHFUZZ to prioritize each seed according to the needs, *i.e.,* reaching new targets in *exploration* or hammering under-tested locations in *explotation* (thus addressing **C3**). In our experiments, we determined these timeouts for each event and leave a discussion for fine-tuning in §7.1. In the rest, we detail the *inter-function* exploration and the *exploitation* phase. For *intra-function* exploration, we re-use the original cull algorithm (for AFL [8] and AFL++ [31]) and thus omit its description.

Inter-function Exploration Phase. In this phase, FISHFUZZ selects seeds to maximize the reached functions containing untriggered 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 $favored = 1$ 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 repeats the process. *getClosestSeedToFun* finds the closest seed s to the function f through a seed-function distance (§3.2). If multiple seeds are equidistant to f, we prefer the one with the lowest execution time.

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

```
1  interFunctionCullQueue(Queue, Functions)
2      for s ∈ Queue do
3          s.favored = 0
4      end
5      for f ∈ Functions do
6          if f.unexplored ∧ f.hasTargets then
7              s ← getClosestSeedToFun(Queue, f)
8              s.favored = 1
9          end
10     end
```

Algorithm 2: Cull logic for the exploitation phase.

```
1  exploitationCullQueue(Queue, Targets)
2      for s ∈ Queue do
3          s.favored = 0
4      end
5      trgs_to_visit ← ∅
6      for t ∈ Targets do
7          if t.reached then
8              trgs_to_visit ← trgs_to_visit ∪ {t}
9          end
10     end
11     trgs_to_visit ← orderByHit(trgs_to_visit)
12     threshold ← |trgs_to_visit| * 20%
13     for (p,t) ∈ enumerate(trgs_to_visit) do
14         if p < threshold then
15             s ← getFastestSeedToTarget(Queue, t)
16             s.favored = 1
17         end
18     end
```

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 the 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 fewest hit targets (line 12 and line 14). For each suitable target, *getFastestSeedToTarget* returns the

fastest seed s that hits t, thus finally setting s as *favored*. As the fastest seed, we select the seed with the lowest execution time, which is measured in modern fuzzers [1, 27]. In this phase, we have a high probability to find seeds that hit targets, which are generated during the exploration phase. Finally, the function *getFastestSeedToTarget* uses the dynamic target priority in §3.1. In our experiments, we select the top 20% of the fewest hit targets, we leave a discussion about parameters' choice in §7.

5 Implementation

We provide two prototypes of FishFuzz: FF_{AFL} and FF_{AFL++}, which are based on AFL [31] version 2.57b and AFL++ [8] version 4.00c, respectively. For the instrumentation, we extended LLVM [19] version 12.0.1.[1]

We implement the *inter-function exploration* and the *exploitation* phases as two cull queue functions in AFL (2,500 LoC) and in AFL++ (1,800 LoC). For the program analysis, we develop additional analysis passes for LLVM to extract CFG, CG, and to 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 are around 200 LoC. As for sanitizers, we use ASan [11, 22] and UBSan [12] distributed with the compiler-rt libraries from LLVM/Clang. We open-source FishFuzz and materials for replicating the experiments at `https://github.com/HexHive/FishFuzz`.

6 Evaluation

We evaluate FishFuzz to answer the following research questions. **RQ1:** How many targets are reached? (§ 6.1) **RQ2:** How efficiently does FishFuzz find bugs? (§ 6.2) **RQ3:** Can FishFuzz find new bugs? (§ 6.3) **RQ4:** Can other fuzzers benefit from our strategies? (§6.4) **RQ5:** How does FishFuzz improve the original fuzzer? (§6.5) We conduct all the experiments by following the best practice described in [24] using the two FishFuzz implementations: FF_{AFL} and FF_{AFL++}. For simplicity, we refer to FishFuzz when indicating the two prototypes, and specify which when needed. In RQ5, we discuss the differences between FF_{AFL} and FF_{AFL++}.

Compared Fuzzers. We test FishFuzz against different families of fuzzers. First, we select three of the most recent DGFs in the literature: TortoiseFuzz [27], ParmeSan [21], and SAVIOR [7]. Then, we include an array of Two-Stage fuzzers: AFLFast [4], FairFuzz [17], EcoFuzz [29], and K-Scheduler [23]. We also include AFL [31] and AFL++ [8] since they are the base for FF_{AFL} and FF_{AFL}, respectively. Moreover, we deploy FishFuzz over QSYM [30] to answer RQ4. For our evaluation, we choose ASan and UBSan as sanitizers as mentioned in §5.

Benchmarks Selected. We choose *four* benchmarks. Two sets of programs come from TortoiseFuzz [27] and SAVIOR [7]. Since we use the TortoiseFuzz benchmark set with the ASan sanitizer, we call it *ASan benchmark*. Likewise, the SAVIOR benchmark contains only UBSan sanitizers, thus we name it *UBSan benchmark*. Regarding the ASan benchmark, it contains 11 programs, *nine* of which from TortoiseFuzz [27],[2] plus `nm-new` and `tcpdump`. Only for ParmeSan, we remove *four* programs from the ASan benchmark due to a non-resolvable exception while instrumenting the targets—while the other fuzzers handle all programs. Regarding the UBSan benchmark, we choose 8 programs from SAVIOR and remove one because it is incompatible with Ubuntu 16.04.[3] For testing the Two-Stage fuzzers, we design an ad-hoc benchmark, called *Two-Stage benchmark*, which covers programs accepting different input formats, *e.g.,* document, image, text, or executable. Finally, we compose a benchmark of 30 real-world programs for the experiments in §6.3. All in all, our evaluation covers 47 programs.

Experiment Setup. All experiments are performed on a Xeon Gold 5218 CPU (22M Cache, 2.30 GHz) equipped with 64GB of memory. We evaluate the ASan and UBSan benchmarks on Ubuntu 16.04 to reproduce the original environment of their respective works. For the two-stage campaign, we opt for Ubuntu 18.04 to provide the same OS to the baseline fuzzers. Finally, we choose Ubuntu 22.04 for real-world programs. All experiments are run in docker containers with one core assigned. For a fair evaluation, we choose the minimized seeds provided by AFL as the initial corpus. If none are present, we leverage the program test cases. Since most fuzzers use a deterministic stage,[4] we do likewise in AFL++. Our artifact contains the full seed corpus and fuzzer configurations.

FishFuzz Hyperparameter Setup. For our evaluation, we fine-tune FishFuzz's hyperparameters for ASan sanitizer (§4.2) since we focus on vulnerabilities, which are commonly found with ASan. For UBSan, we conduct a preliminary investigation using the same hyperparameters, and discuss their tuning in §7. Therefore, we use the same set of hyperparameters for ASan, UBSan, and Two-Stage benchmarks. Specifically, we set 30 min for *inter-function exlporation*, 10 min for *intra-function exploration*, and 1 hour for *exploitation* (§4.2).

[1] We also successfully test it on LLVM 10.0.1

[2] We discard `libming` due to incompatibility with clang-12

[3] `objdump` runs out of memory during compilation.

[4] `https://afl-1.readthedocs.io/en/latest/about_afl.html`

6.1 RQ1: How many targets are reached?

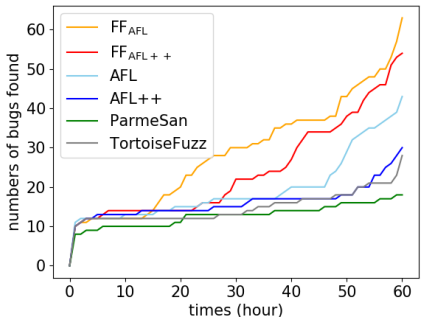
We evaluate if the *exploration phase* of FISHFUZZ can reach more targets with respect to its competitors. To this end, we set the following experiments. First, we exercise ParmeSan and TortoiseFuzz against the ASan benchmark, SAVIOR against the UBSan benchmark, and the Two-Stage fuzzers against their benchmark. Then, we evaluate FISHFUZZ against the ASan, UBSan, and Two-Stage benchmarks. For ASan, we run 10 rounds 60 hours each, while for UBSan and Two-Stage, we run 10 rounds 24 hours each (as in the original papers [7, 27]). The results for ASan, UBSan, and Two-Stage benchmarks are in Table 3, Table 4, and Table 5, respectively, along with a dedicated analysis of p-values in §A.2.

Overall, Table 3, Table 4, and Table 5 show FISHFUZZ's prototypes achieve higher coverage in all three benchmarks. Specifically, FF_{AFL++} achieves better coverage in the ASan and Two-Stage benchmarks (+11.69% and +2.87% compared against the second best), while FF_{AFL} has better performance with UBSan (+1.99% compared against the second best). For the UBsan benchmark, we observe that AFL++ and SAVIOR reach better coverage for 3 programs (`djpeg`, `tiff2pdf`, and `xmllint`). We look closely at these cases and conclude that UBSan's targets are easier to reach compared to ASan's ones. This can be observed by looking at each fuzzers' performances (Table 4), they have similar results for almost all the programs. However, when considering the aggregate results, FISHFUZZ's prototypes achieve higher coverage. These results also reflect in the p-values (Table 11), which are not statistically significant for the UBSan benchmark. The only exceptions are `tcpdump` and `readelf`. We remark that, even though FISHFUZZ is not fine-tuned for UBSan, FF_{AFL} obtains similar or better performances than its competitors (see § 7). We further investigate the incongruence between the results of SAVIOR, ParmeSan, and TortoiseFuzz compared to their papers [7, 21, 27]. For SAVIOR and TortoiseFuzz, the original papers assign 3 cores to each fuzzing campaign. Conversely, we assign one core each to provide the same amount of resources to each fuzzer. ParmeSan does not mutate the initial seed length and uses custom seeds in their evaluation, while we choose the initial corpus from AFL's test cases. For AFL++, we observe the deterministic stage sometimes reduces efficiency, thus resulting in worse performance than AFL, as also observed by Wu et al. [28].

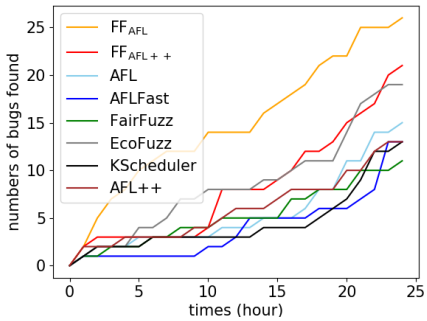
Takeaway: Our experiment shows that the *exploration phase* of FISHFUZZ reaches more targets compared to modern fuzzers and with similar, if not better, results of their baseline.

6.2 RQ2: How efficiently does FISHFUZZ find bugs?

We evaluate the ability of the *exploitation* phase in triggering bugs. To this end, we choose the ASan, UBSan, and Two-Stage benchmarks. Similar to §6.1, we exercise 10 rounds



(a) Time-To-Exposure for ASan Evaluations.



(b) Time-To-Exposure for Two-Stage Evaluations.

Figure 5: Time-To-Exposure for ASan and Two-Stage benchmarks.

of 60 hours for ASan, while 24 hours per run for UBSan and Two-Stage (as in the original paper [7, 21]). Then, we measure the number of unique bugs for ASan and Two-Stage and the number of triggered targets in UBSan. For ASan/Two-Stage, we report the number of unique bugs because each bug can be associated with multiple targets, thus it could be ambiguous simply referring to the targets. Conversely, UBSan targets might not be associated with 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 maintainer, and thus not considered a bug.

For ASan/Two-Stage, we identify unique bugs by first hashing stack traces to disambiguate crashes, followed by manually triaging bugs. For UBSan, we extract the output patterns and identify their source locations (as done by SAVIOR's authors after contacting them). In our evaluation, we observe that ASan targets require, on average, more hits than UBSans targets before triggering a crash. We thus consider UBsan's targets simpler, § A.1 provides a detailed discussion about ASan and UBSan targets.

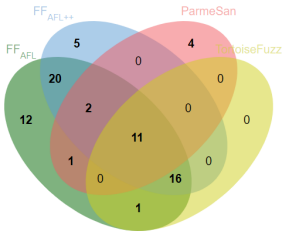
Bugs Found. The results for the ASan, UBSan and Two-Stage benchmarks are shown in Table 6, Table 4, and Table 7, respectively. Overall, FISHFUZZ triggers more bugs/targets

Table 3: Edge coverage and targets reached in the ASan benchmark. The results refer to the average of 10 rounds for 60 hours each.

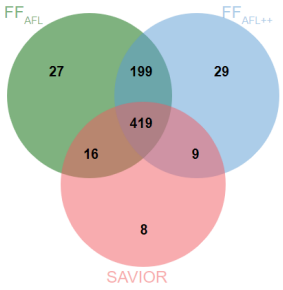
	FF_{AFL}		FF_{AFL++}		AFL		AFL++		ParmeSan		TortoiseFuzz	
	cov	reach	cov	reach	cov	reach	cov	reach	cov	reach	cov	reach
exiv2	19373.9	9524.1	**21199.1**	**10168.6**	11140.9	6764.0	19889.6	9676.5	-	-	8612.5	5888.0
flvmeta	996.0	152.0	994.6	151.7	995.2	151.8	996.0	152.0	961.8	150.0	994.2	152.0
MP4Box	**13751.1**	**1663.4**	11580.6	1371.3	11524.2	1405.0	9320.7	1215.5	10411.0	1160.8	9855.0	1238.9
lou_checktable	2605.6	242.4	**2816.3**	**305.4**	2620.4	240.7	2131.9	187.5	1760.2	138.9	2392.7	214.1
tiff2pdf	**18517.6**	3484.9	18466.1	**3494.9**	18047.8	3337.1	17919.5	3361.5	12069.6	2436.7	15850.9	2977.7
nasm	11295.3	1860.6	**12048.9**	**2018.3**	10974.5	1751.7	11541.7	1889.7	8441.9	1489.9	10275	1698.4
tcpprep	**1042.7**	**212.2**	961.1	187.7	1030.6	209.0	1037.8	211.9	-	-	995.8	209.5
catdoc	**707.0**	**121.0**	418.0	80.0	705	120.6	418.0	80.0	-	-	682.6	119.1
tcpdump	26389.8	5924.0	**29601.9**	**6765.9**	25010.1	5187.6	23048.8	5131.8	-	-	16667.4	3364.2
nm-new	15458.7	3731.4	**15632.7**	**3788.7**	13998.7	3237.7	15430.4	3755.9	9505.3	2709.4	13223.3	3123.3
gif2tga	**747.1**	**132.4**	743.9	132.3	728.1	123.9	745.2	132.8	722.9	125.7	735.2	126.6
sum	110884.8	27048.4	**114463.2**	**28464.8**	96775.5	22529.1	102479.6	25795.1	43872.7	8211.4	80284.6	19111.8

Table 4: Edge coverage, targets reached and triggered in UBSan benchmark. The results refer to the average of 10 rounds for 60 hours each.

	FF_{AFL}			FF_{AFL++}			AFL			AFL++			SAVIOR		
	cov	reach	trigger	cov	reach	trigger	cov	reach	trigger	cov	reach	trigger	cov	reach	trigger
djpeg	12285.8	3940.6	138.2	12243.7	3953.2	137.8	11846.9	3866.1	134	**12490.6**	**3987.7**	**141.9**	11,919.9	3874.3	134.4
jasper	**10708.1**	**1751.4**	48.9	10158.6	1639.3	37.9	10313.8	1682.2	36.7	10553	1708.8	**52.7**	10,472.0	1664.7	44.5
objdump	**10128.3**	**1092.7**	**95.8**	9084.4	1123	90.7	9800.1	1040.2	94.9	9430.5	995.6	92.2	-	-	-
readelf	**2041.9**	**163.7**	**28.3**	1852.6	141.7	23	1895.6	148.3	21.8	1947.1	152.4	26.7	2,097.9	173.6	22.9
tcpdump	24615.0	3863.6	152.6	**24929**	**4033.1**	**154.7**	24198.9	3709.3	143.9	22553.9	3472	134.1	17,690.6	2752	93.5
tiff2pdf	13160.1	1932.9	14.2	13004.9	1885.2	15.3	**13237.9**	**1935.6**	**16.2**	12985.6	1908.8	13.6	12,611.7	1816.9	12.5
tiff2ps	**9073.4**	**1172.1**	**14.9**	9007.4	1160.1	13.6	9048.9	1165	13.5	8772.2	1118.5	10.5	8,765.1	1126.9	11.4
xmllint	8029.3	722.3	15.2	8048.7	718.8	**15.9**	7940.2	693.6	12.9	**8265.9**	**736.7**	15.1	7,849.9	711.2	13.9
sum	**90041.9**	14639.3	**508.1**	88329.3	**14654.4**	488.9	88282.3	14240.3	473.9	86998.8	14080.5	486.8	71,407.0	12119.6	333.1



(a) Unique bugs found in the ASan benchmark and compared with FISHFUZZ, TortoiseFuzz, and ParmeSan.



(b) Triggered targets in the UBSan benchmark and compared with FISHFUZZ and SAVIOR.

Figure 6: Bugs found and triggered targets.

compared to the state-of-the-art. Specifically, both FF_{AFL} and FF_{AFL++} find 40.1 and 38 unique bugs on average in the ASan benchmark (Table 6), doubles than either AFL++ (18.9), TortoiseFuzz (18), or ParmeSan (14.2). We observe a similar pattern for the Two-Stage benchmark (Table 7), where FISHFUZZ prototypes find more bugs than its competitors on average (from 57.89% to 346.81% more unique bugs). Here, the only exception is EcoFuzz, which can find slightly more unique bugs than FF_{AFL++} (3.01%). Most importantly, both FF_{AFL} and FF_{AFL++} trigger more bugs compared to their baseline fuzzers (*i.e.,* AFL and AFL++) in both ASan and Two-Stage benchmarks, thus showing FISHFUZZ improves the bugs finding capabilities in the original fuzzers. For the UBSan benchmark, we observe FF_{AFL} (508.1) triggers more targets compared to SAVIOR (333.1), while FF_{AFL++} achieves the second-best result (488.9). In particular, we activate from 2.83% to 52.54% more targets than SAVIOR on average. UBSan's results suffer from a similar problem in terms of coverage, since UBSan's targets are simpler compared to ASan, the solely *exploration* triggers fewer targets (more details in §A.1). In the current evaluation, we test UBSan with the same tuning as ASan, thus we give higher importance to *exploitation* rather than *exploration*. Conversely, hyperparameters that prioritize *exploration* produce better results for UBSan. §7.1 shows a discussion of hyperparameters.

Time-To-Exposure. We measure the Time-to-Exposure for the ASan and Two-Stage benchmarks. Similar to the bug report, we only consider the time to exposure for the ASan and Two-Stage benchmarks whose bugs are uniquely identifiable with an ID. For comparison, we consider the lowest

Table 5: Edge coverage and targets reached in the Two Stage Evaluation benchmark. The results refer to the average of 10 rounds for 24 hours each.

program	FF_{AFL}		FF_{AFL++}		AFL		AFLFast		FairFuzz		EcoFuzz		K-Scheduler		AFL++	
	cov	reach	cov	reach	cov	reach	cov	reach	cov	reach	cov	reach	cov	reach	cov	reach
cflow	3974.3	999.5	**3979.2**	**1000.8**	3963.6	991.1	3952.0	988.5	3940.1	985.0	3964.8	997.1	3938.9	985.0	3954.2	989.2
mujs	6474.4	1558.3	**6596.6**	**1594.8**	6009.2	1469.9	6122.8	1470.1	6299.4	1520.0	6156.5	1482.3	6022.5	1451.4	6433.6	1547.9
tic	4731.4	925.5	**4819.3**	**944.2**	4135.4	785.5	4271.1	823.6	4469.9	864.2	4550.9	881.6	4272.5	807.9	4586.7	888.9
mutool	**8367.7**	**2158.5**	8298.8	2138.5	8132.1	2093.1	8231.4	2100.1	8268.8	2112.6	8215.7	2097.7	-	-	8254.7	2104.4
dwarfdump	**3091.1**	**730.1**	2973.9	704.1	2999.8	698.9	2789.5	641.9	2966.1	690.0	3005.7	707.7	2940.9	695	2967.0	702.9
w3m	3605.4	822.2	**3695.7**	**828.4**	2822.7	730.0	2833.0	734.0	2836.0	734.0	3655.6	821.9	2963.3	748.3	3646.9	817.6
cxxfilt	7700.4	1685.8	**8027.8**	**1777.9**	6843.0	1350.0	7062.2	1416.4	7154.0	1436.3	6968.8	1393.2	6875.5	1369.7	7476.0	1513.4
sum	37944.7	8879.9	**38391.3**	**8988.7**	34905.8	8118.5	35262	8174.6	35934.3	8342.1	36518	8381.5	27013.6	6057.3	37319.1	8564.3

Table 6: Unique bugs found in the ASan benchmark after 10 rounds of 60 hours each.

	FF_{AFL}	FF_{AFL++}	AFL	AFL++	ParmeSan	TortoiseFuzz
tcpdump	0.6	0.8	0.7	0.2	-	0
catdoc	0.9	0	1.0	0	-	0
exiv2	1.3	0.8	0	0.7	-	0
flvmeta	2.0	2.0	2.0	2	1.7	2.0
lou_checktable	1.9	3.0	0.4	0	0	0.1
nasm	3.3	3.1	2.2	2.6	0	1.1
nm-new	6.6	6.7	1.6	0.1	0	0.6
tcpprep	1.8	2.0	2.0	1.8	-	2.0
tiff2pdf	0.3	0.1	0	0.2	0	0.2
gif2tga	4.0	4.0	4.0	4.0	4.0	4.0
MP4Box	17.4	15.5	14.8	7.3	8.5	8.0
sum	40.1	38.0	28.7	18.9	14.2	18.0
sum (- MP4Box)	22.7	22.5	13.9	11.6	5.7	10.0
vs FF_{AFL}			+39.72%	+112.17%	+182.39%	+122.78%
vs FF_{AFL++}			+32.40%	+101.06%	+167.61%	+111.11%

time-to-exposure between FF_{AFL} and FF_{AFL++}. Our results show that, out of 104 known bugs, FISHFUZZ finds 71 of them faster than previous work (68.3%), 8 in the same time (7.7%), and only 25 took slightly longer (24.0%). §A.5 shows detailed data for all the bugs found. Additionally, Figure 5 shows the bug discovery over time. The images show FF_{AFL} and FF_{AFL++} are almost always above their competitors, especially at the end of the fuzzing session. Only EcoFuzz seems competitive with FF_{AFL++}. Most importantly, FISHFUZZ's prototypes always outperform their baseline (*i.e.,* AFL and AFL++). FISHFUZZ finds 38 new bugs not previously discovered, we discuss them in a dedicated section in §6.3.

Bugs Analysis. Finally, we analyze the overlap of unique bugs and triggered targets found by FISHFUZZ and its competitors. For this analysis, we adopt the following procedures. First, we ignore AFL and AFL++, which receive a dedicated analysis in §6.5. Then, we use two Venn Diagrams for ASan and UBSan in Figure 6a and Figure 6b, since the results are suitable for this demonstration. For the Two-Stage benchmark, instead, we use Table 9, which contains more detailed overlap information for each combination of fuzzers in that benchmark. For ASan (Figure 6a), FF_{AFL} and FF_{AFL++} share 93 and 75 unique bugs with ParmeSan and TortoiseFuzz. We did not manage to find only 4 for ParmeSan and one for AFL++, belonging to `gpac` and `mutool` respectively. For UBSan (Figure 6b), instead, we triggered 707 targets in total, while 8 targets were triggered by SAVIOR only. After investigation, we notice that FF_{AFL} and FF_{AFL++} do not reach the missing targets. Since SAVIOR is based on symbolic execution, we thus conclude SAVIOR and FISHFUZZ share a fundamentally different exploration phase that reaches different code sections. This leaves room for future alternative *exploration* phases in FISHFUZZ.

Table 9 shows the overlapping bugs found in Two-Stage benchmark. Since this benchmark contains too many fuzzers for a useful Venn Diagram, we opt for a table whose cells indicate two overlap measurements for each combination of fuzzers. Specifically, given a pair of fuzzers, we indicate the overlap with two numbers: *a* and *b*; *a* is the number of bugs found *exclusively* by the two fuzzers (but not others), while *b* is the number of bugs found by *both* fuzzers (and other fuzzers). For example, the intersection between FF_{AFL} and FF_{AFL++} shows 7 and 20: 7 bugs are found *exclusively* by FF_{AFL} and FF_{AFL++}, while 20 bugs are found by FF_{AFL} and FF_{AFL++} and other fuzzers. We also note Table 9 is mirrored since *a* and *b* are reflexive (*i.e., a* and *b* are not influenced by the pair order). Overall, we notice that FF_{AFL} and FF_{AFL++} have a large intersection with the other competitors, specifically FF_{AFL} is significantly bigger than their respective baseline. This means FF_{AFL} finds the same bugs as the other competitors and more bugs than their original versions (*i.e.,* AFL). For instance FF_{AFL}'s row has the largest *b* value among other combinations of fuzzers, showing it can cover a large number of bugs. FF_{AFL++}, instead, has a similar overlap as EcoFuzz. We conduct a dedicated study about FF_{AFL} and FF_{AFL++} respect AFL and AFL++ in §6.5.

Takeaway: Our experiments show the ability of our *exploitation* phase to discover more bugs and trigger more targets compared to the state-of-the-art. Additionally, we show that FISHFUZZ finds known bugs faster.

6.3 RQ3: Can FISHFUZZ find new bugs?

We challenge the ability of FISHFUZZ to find new CVEs in real applications. For this experiment, we choose 30 pro-

Table 7: Unique bugs found in Two-Stage benchmark after 10 rounds of 24 hours each.

	FF_{AFL}	FF_{AFL++}	AFL	AFL++	AFLFast	FairFuzz	EcoFuzz	KScheduler
cflow	2.0	2.3	0.8	1.5	1.0	1.7	1.5	1.7
mujs	4.5	4.5	0.4	1.2	0.6	0.8	1.4	0.3
tic	3.4	3.9	0.9	1.7	0.7	1.1	0.7	0.1
mutool	1.0	0.4	0.7	0.9	0.5	1.3	1.1	-
dwarfdump	2.0	1.7	2.0	1.6	1.5	2.0	1.9	2.0
w3m	0	0	0	0	0	0	0	0
cxxfilt	8.1	0.1	3.2	0.2	0.4	0	6.7	1.8
sum	21.0	12.9	8.0	7.1	4.7	6.9	13.3	5.9
vs FF_{AFL}			+162.50%	+195.77%	+346.81%	+204.35%	+57.89%	+255.93%
vs FF_{AFL++}			+61.25%	+81.69%	+174.47%	+86.96%	-3.01%	+118.64%

grams from top tiers publications, *i.e.,* TortoiseFuzz [27], SAVIOR [7], GREYONE [9], FuzzGen [14], as well as from the fuzzing community. For each program, we deployed ASan and UBSan, respectively. We run a session one week long for each program.

In total, we found 56 new bugs, 38 of which were confirmed CVEs. FISHFUZZ finds most of the bugs/CVEs in less than three days (46), while 10 bugs require from three to seven days. We found 44 bugs with the ASan sanitizers and 12 bugs with UBSan. The most common bugs are heap-overflow (15), reachable assertions (7), stack-exhausted (6), divide-by-zero (2), and shift exponentially (1). We report the full list of bugs founds in our artifact. Most importantly, the bugs found are from exhaustively fuzzed programs (*i.e.,* programs intensively tested by other state-of-the-art fuzzers). For instance, FISHFUZZ finds CVE-2022-27943 in `nm-new` (`binutils`), one of the most widely used fuzzing test suites. Moreover, FISHFUZZ also finds *five* bugs in Android and Apple OS libraries (`libavc`, `libmpeg2`, `liblouis`), *two* of which (CVE-2022-26981, CVE-2022-31783) have been confirmed to enable remote code execution and receive acknowledgments from Apple and Huawei (EulerOS-SA-2022-2226, HT213340).

Takeaway: FISHFUZZ is effective in finding new CVEs since it manages to find 38 new vulnerabilities in less than a week over programs already deeply tested by previous works.

6.4 RQ4: Can other fuzzers benefit from our strategies?

To answer this question, we combine QSYM [30] with FF_{AFL} and AFL to measure if our cull queuing improves the performance. Specifically, we run QSYM with one AFL-primary and one concolic executor (more details in QSYM original paper [30]). We run the experiments against UBSan benchmark for 10 rounds of 24 hours each.

Table 8 shows that combining QSYM+FISHFUZZ improves every aspect of the original fuzzer. For instance, we improve the triggered targets up to 33.29% compared to QSYM+AFL and up to 21.21% and 14.60% for targets reached and coverage, respectively. Finally, FISHFUZZ also improves the number of seeds in the queue by reaching 41.25% more seeds at maximum (*i.e.,* path column).

Takeaway: This experiment demonstrates that FISHFUZZ is compositional and improves the performance of other fuzzers.

6.5 RQ5 - How FISHFUZZ improve the fuzzer capabilities?

FISHFUZZ is a generic input prioritization technique that can be deployed over any fuzzer. The purpose of FISHFUZZ is to select better seeds for *exploration* or *exploitation*, without altering the original fuzzer mutators or other mechanisms. Therefore, the performance of FISHFUZZ might change depending on the originally chosen fuzzer. In this section, we study these differences by looking at how FF_{AFL} and FF_{AFL++} improve over their baselines AFL and AFL++.

Looking at Table 3 and Table 5, we observe that FF_{AFL} and FF_{AFL++} cover more edges compared to their baselines (13.0% and 9.3%), as well as reach more targets (17.2% and 9.0%). Similar differences also appear when comparing single programs, for `exiv2` (ASan benchmark), FF_{AFL++} and AFL++ achieve significantly better coverage compared to FF_{AFL}/AFL (9.4% and 78.5% respectively). However, in `catdoc`, FF_{AFL++}/AFL++ perform worse than FF_{AFL}/AFL (about 40.8% lower).

More interestingly, bugs covered by FF_{AFL++} and FF_{AFL} are different. Since FISHFUZZ does not modify the original queue culling for *intra-function exploration*, FF_{AFL} and FF_{AFL++} show different results for bug discovery. For example, in Table 7, AFL++ does not perform well in `cxxfilt` (0.2 bugs on average) while AFL discovers 3.2 bugs. In Two-Stage, FF_{AFL} and AFL cover the same seven bugs, however, CVE-2017-13731 is only covered by FF_{AFL++}/AFL++. In ASan, we observe similar cases: `gpac_issue_1250` and `catdoc_issue_8` are only covered by AFL/FF_{AFL}. For all

Table 8: Running QSYM+FF$_{AFL}$ against QSYM+AFL in the UBSan benchmark. The results refer to the average of 10 rounds for 24 hours each. Column path represents the number of seeds in the queue.

	QSYM+AFL				QSYM+FF$_{AFL}$				vs QSYM+FF$_{AFL}$			
	path	cov	reach	trigger	path	cov	reach	trigger	path	cov	reach	trigger
djpeg	1315.9	10640.0	3435.4	91.7	2264.5	11964.9	3905.0	138.1	+72.09%	+12.45%	+13.67%	+50.60%
jasper	1029.7	9563.0	1430.1	29.0	1596.5	10565.7	1687.3	40.8	+55.05%	+10.49%	+17.98%	+40.69%
readelf	516.8	2470.8	220.9	20.1	654.7	2517.6	218.6	23.8	+26.68%	+1.89%	-1.04%	+18.41%
objdump	1812.8	9223.6	968.0	60.6	2142.7	10083.4	1112.4	78.6	+18.20%	+9.32%	+14.92%	+29.70%
tcpdump	1760.1	13010.3	1981.8	73.7	2470.5	16924.3	2626.8	93.0	+40.36%	+30.08%	+32.55%	+26.19%
tiff2pdf	1258.5	9131.7	993.2	12.0	1806.8	9878.1	1144.1	12.4	+43.57%	+8.17%	+15.19%	+3.33%
tiff2ps	815.2	6355.7	540.2	6.7	1594.6	8221.0	1022.7	7.1	+95.61%	+29.35%	+89.32%	+5.97%
xmllint	2272.2	8112.0	691.0	10.5	2698.5	8355.3	720.1	11.8	+18.76%	+3.00%	+4.21%	+12.38%
sum	10781.2	68507.1	10260.6	304.3	15228.8	78510.3	12437.0	405.6	+41.25%	+14.60%	+21.21%	+33.29%

these cases, we leave detailed tables in §A.5.

Takeaway: FISHFUZZ enhances the capabilities of the original fuzzers by directing the energy toward promising locations. More importantly, FISHFUZZ does not change the *intra-function exploration*, thus triggering different bugs.

7 Discussion

We discuss hyperparameters tuning (§7.1), target size (§7.2), and combination with orthogonal techniques (§7.3).

7.1 Hyperparameter Tuning

FISHFUZZ uses a finite-state machine to alternate *exploration* and *exploitation*, whose switching is governed by hyperparameter (§4.2). In our experiments, we empirically try different settings and observe that a long *exploitation* time (*i.e.,* 1 hour) leads to better performances for ASan. However, this setting brings to non-optimal performance for UBSan. We argue this is because UBSan's targets are simpler, thus they require less *exploitation* time. To validate our hypothesis, we conduct an additional study in §A.3. The results suggest different hyperparameters can improve up to 12.03% coverage and 37.29% triggered targets. To sum up, different scenarios require their own set of hyperparameters, likewise for ML/DL [33],

7.2 Target Set Size

FISHFUZZ is designed to handle large target sets (up to 20*k* in our experiments §6). 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 [20]. Second, we believe this observation suggests the need for 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.

7.3 Combining FISHFUZZ with other works

BEACON [13], 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 combining SAVIOR [7] with our *exploration* phase to investigate if different approaches can lead to better performances.

8 Related Works

FISHFUZZ improves existing fuzzing work across two research areas: Directed Greybox Fuzzers (§8.1) and Two-Stage Fuzzers (§8.2).

8.1 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 [3], which models the seed-target distance as a harmonic average distance. However, the AFLGo approach loses 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 [5] and Peiyuan et al. with FuzzGuard [33]. 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 analysis to resolve indirect jumps, while its seed selection automatically promotes interesting inputs.

Steps toward more scalable DGF are discussed by Österlund with ParmeSan [21] and Chen with SAVIOR [7], respectively. Both ParmeSan and SAVIOR consider all the sanitizer labels as targets. Additionally, SAVIOR introduces a heavy reachable analysis to select interesting inputs. Both works suffer from the original AFLGo limitation since they collapse

Table 9: Unique bug finding overlap for the Two-Stage benchmark. For each pair of fuzzers, we indicate two overlap measurements a/b; where a indicates the bugs found *exclusively* by both fuzzers, while b indicates the bugs found by both fuzzers but possibly also others. As an example, the intersection FF_{AFL} and FF_{AFL++} indicates $7/20$: 7 is the number of bugs found *exclusively* by FF_{AFL} and FF_{AFL++}, while 20 are bugs found by FF_{AFL} and FF_{AFL++} and by other fuzzers as well. The table is mirrored because a and b are reflexive (the pair order does not matter).

	FF_{AFL}	FF_{AFL++}	AFL	AFL++	AFLFast	FairFuzz	EcoFuzz	KScheduler
FF_{AFL}	2/29	7/20	0/14	0/11	0/13	0/10	1/19	0/13
FF_{AFL++}	7/20	0/21	0/9	1/12	0/9	0/10	0/12	0/8
AFL	0/14	0/9	0/15	0/10	0/11	0/9	0/15	0/11
AFL++	0/11	1/12	0/10	0/13	0/8	0/10	0/11	0/7
AFLFast	0/13	0/9	0/11	0/8	0/13	0/9	0/13	0/11
FairFuzz	0/10	0/10	0/9	0/10	0/9	0/11	0/11	0/7
EcoFuzz	1/19	0/12	0/15	0/11	0/13	0/11	0/20	0/13
KScheduler	0/13	0/8	0/11	0/7	0/11	0/7	0/13	0/13

the seed-target distance into a scalar, thus losing precision. FISHFUZZ differs from these works for two reasons. First, it employs a novel seed-target distance that overcomes scalability limitations. Second, it uses a faster exploitation phase to trigger more targets.

Lee et al. propose CAFL [16] (Constraint guided directed greybox fuzzing). The goal of this work is to synthesize a POC from a given crash by following a similar approach to AFLGo. In their scenario, CAFL considers only one target, while FISHFUZZ is designed to handle a large number of targets. Finally, Zhu et al. discuss Regression Greybox Fuzzing [32], 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 input prioritization strategy, while they recognize interesting code locations for testing.

Huang et al. introduce BEACON [13], 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.

8.2 Two-Stage Fuzzers

Two-Stage fuzzers use exploration and exploitation phases to reach and trigger multiple targets. Böhme et al. proposes AFLFast [4], which relies on Markov chains to probabilistically select seeds that improve the coverage. Their contribution is mainly energy distribution related, while queue culling and seed distance are not discussed.

Lemieux et al. [17] study new mutation strategies and seed selections to hit rare branches. Their contribution is more related to improving code coverage, while FISHFUZZ also maximizes the targets triggered. Yue et al. discuss a combination of adaptive energy schedule and game theory to avoid testing unfruitful seeds. Their approach does not discuss seed selection strategies, thus being orthogonal to FISHFUZZ.

Wang et al. [27] 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 adjusting the fuzzer's energy toward more promising code locations. Conversely, the queue culling mechanism of FISHFUZZ is adaptive and can be potentially used to improve the performance of Wang's work.

9 Conclusion

Exploration and *exploitation* are fuzzers' key components for finding bugs. Greybox fuzzers overly focus on *explorations*. 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 priorities 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 47 programs and have, so far, discovered 56 new bugs (38 CVEs). FISHFUZZ is available at https://github.com/HexHive/FishFuzz and we provide a test environment to play with our novel input prioritization mechanism.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their feedback. This work was supported, in part, by the National Natural Science Foundation of China (U1836210), the Key Research and Development Science and Technology of Hainan Province (GHYF2022010), the China Scholarship Council, the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), SNSF PCEGP2_186974, and DARPA HR001119S0089-AMP-FP-034.

References

[1] AFL. technical_details.txt. https://github.com/google/AFL/blob/master/docs/technical_details.txt, 2019.

[2] Domagoj Babic, Stefan Bucur, Yaohui Chen, Franjo Ivancic, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei 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.

[3] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[5] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2095–2108, 2018.

[6] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[7] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1580–1596. IEEE, 2020.

[8] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[9] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. {GREYONE}: Data flow sensitive fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2577–2594, 2020.

[10] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: Whitebox fuzzing for security testing: Sage has had a remarkable impact at microsoft. *Queue*, 10(1):20–27, 2012.

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

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

[13] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Beacon: Directed grey-box fuzzing with provable path pruning.

[14] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. {FuzzGen}: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2271–2287, 2020.

[15] Donald B Johnson. A note on dijkstra's shortest path algorithm. *Journal of the ACM (JACM)*, 20(3):385–388, 1973.

[16] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. Constraint-guided directed greybox fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3559–3576. USENIX Association, August 2021.

[17] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485, 2018.

[18] Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos. Lock-in-pop: Securing privileged operating system kernels by keeping on the beaten path. In *USENIX Annual Technical Conference*, pages 1–13, 2017.

[19] llvm. The LLVM Compiler Infrastructure Project. http://llvm.org/.

[20] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX*

Security Symposium (USENIX Security 19), pages 1949–1966, 2019.

[21] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. {ParmeSan}: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2289–2306, 2020.

[22] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

[23] Dongdong She, Abhishek Shah, and Suman Jana. Effective seed scheduling for fuzzing with graph centrality analysis. *arXiv preprint arXiv:2203.12064*, 2022.

[24] Erik van der Kouwe, Gernot Heiser, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. SoK: Benchmarking Flaws in Systems Security. In *EuroS&P*, June 2019.

[25] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High system-code security with low overhead. In *2015 IEEE Symposium on Security and Privacy*, pages 866–879. IEEE, 2015.

[26] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu 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)*, pages 1–15, 2019.

[27] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In *NDSS*, 2020.

[28] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. One fuzzing strategy to rule them all. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1634–1645, 2022.

[29] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. {EcoFuzz}: Adaptive {Energy-Saving} greybox fuzzing as a variant of the adversarial {Multi-Armed} bandit. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2307–2324, 2020.

[30] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, Baltimore, MD, August 2018. USENIX Association.

[31] Michal Zalewski. american fuzzy lop. https://lcamtuf.coredump.cx/afl/, 2013.

[32] Xiaogang Zhu and Marcel Böhme. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2169–2182, 2021.

[33] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2255–2269. USENIX Association, August 2020.

A Appendix

A.1 UBSan vs ASan targets

We observe that UBSan targets tend to require less exploitation time compared to ASan targets. Therefore, we conduct a dedicated study regarding the difficulties to trigger targets in the two sanitizers. First, we instrument the Two-Stage benchmark with ASan and UBSan respectively (more info in §6 "Benchmarks Selected"). Then, we count the visit frequencies of targets before triggering a crash and report the average. We observe that in most programs, ASan's average hit rate is significantly higher than UBSan's. For instance, in `cxxfilt` and `mujs`, ASan's hit counts are 1.23*X* and 1.40*X* higher than UBSan's. In `mutool` and `tic`, instead, the differences are more pronounced, *i.e.,* ASan requires 23.17*X* and 1369.87*X* higher hit rates than UBSan. Observing these results, we conclude that UBSan targets are simpler, thus requiring less exploitation time. Our hyperparameter study observes similar results in §7.1.

A.2 P-Values

Table 10, Table 11, and Table 12 contain the p-values from the Mann-Whitney U test. The numbers are the respective p-values from the Table 3, Table 4, and Table 5. We compute the p-values for coverage, target reached/triggered in all our benchmarks by using scipy version 1.8.0 and the results from FF_{AFL++} as baseline. We chose FF_{AFL++} as baseline, instead of FF_{AFL}, because the latter produces unexpected results due to an anomality with the statistical test if all values of one target are equal. For instance, even though FF_{AFL} and TortoiseFuzz achieve similar coverage on `flvmeta`, considering FF_{AFL++} as baseline produces a p-value of 0. This happens because FF_{AFL} explores *exactly* 996 *edges every round*, so the probability for FF_{AFL} to achieve different edge coverage is 0. TortoiseFuzz, instead, explores 995 and 994 edges during the 10 rounds. Conversely, using FF_{AFL++} as baseline procudes more representative p-values. For completeness, we release all the raw measurement data along

Table 10: p-values of the Mann-Whitney U test from the experiments in Table 3.

programs	FF_{AFL} cov	FF_{AFL} reach	FF_{AFL++} cov	FF_{AFL++} reach	AFL cov	AFL reach	AFL++ cov	AFL++ reach	TortoiseFuzz cov	TortoiseFuzz reach	ParmeSan cov	ParmeSan reach
exiv2	0.0046	0.0028	1	1	0.0004	0.0004	0.2123	0.1405	0.0002	0.0002	0.0001	0.0001
flvmeta	0.1681	0.1681	1	1	0.6264	0.6264	0.1681	0.1681	0.017	0.1681	0.0001	0.0001
MP4Box	0.0073	0.0004	1	1	0.7337	0.3845	0.0002	0.0002	0.001	0.001	0.1212	0.001
lou_checktable	0.1857	0.0015	1	1	0.1855	0.0004	0.0045	0.0002	0.0257	0.0002	0.0002	0.0001
tiff2pdf	0.6232	1	1	1	0.0036	0.0006	0.0022	0.0017	0.0002	0.0002	0.0002	0.0002
nasm	0.0002	0.0004	1	1	0.0002	0.0002	0.0008	0.0006	0.0002	0.0002	0.0002	0.0002
tcpprep	0.0007	0.0085	1	1	0.0689	0.4425	0.0001	0.0226	0.1397	0.8755	0.0001	0.0001
catdoc	0	0	1	1	0.0001	0	1	1	0.0001	0	0	0
tcpdump	0.0006	0.0008	1	1	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	0.0001	0.0001
nm-new	0.162	0.1405	1	1	0.0002	0.0002	0.5708	0.7912	0.0002	0.0002	0.0002	0.0002
gif2tga	0.0659	0.8236	1	1	0.0002	0.0001	0.7592	0.5171	0.0012	0.0001	0.0006	0.0008

Table 11: p-values of the Mann-Whitney U test from the experiments in Table 4.

programs	FF_{AFL} cov	FF_{AFL} reach	FF_{AFL} trigger	FF_{AFL++} cov	FF_{AFL++} reach	FF_{AFL++} trigger	AFL cov	AFL reach	AFL trigger	AFL++ cov	AFL++ reach	AFL++ trigger	SAVIOR cov	SAVIOR reach	SAVIOR trigger
djpeg	0.6232	0.3603	0.6998	1	1	1	0.0046	0.0013	0.0344	0.4274	0.6772	0.1333	0.0091	0.0045	0.1003
jasper	0.0058	0.0282	0.0045	1	1	1	0.162	0.4053	0.5959	0.0173	0.1301	0.0004	0.064	0.1617	0.405
readelf	0.0013	0.0028	0.0004	1	1	1	0.1403	0.185	0.4253	0.0757	0.0818	0.1081	0.001	0.0019	0.0773
objdump	0.65	0.273	0.6429	1	1	1	0.0452	0.0211	0.7018	0.0058	0.0022	0.7026	0.0001	0.0001	0.0001
tcpdump	0.1212	0.1041	0.9396	1	1	1	0.0539	0.0211	0.0489	0.0036	0.0019	0.0002	0.0002	0.0002	0.0002
tiff2pdf	0.5205	0.5706	0.7885	1	1	1	0.3075	0.3071	0.0108	0.6232	0.8205	0.1153	0.0257	0.1405	0.0224
tiff2ps	0.2119	0.13	0.298	1	1	1	0.0539	0.1859	0.8479	0.0006	0.0005	0.0056	0.0113	0.0058	0.0458
xmllint	0.4055	0.3432	0.0198	1	1	1	0.9097	0.009	0.0021	0.6776	0.0491	0.0147	0.0003	0.0021	0.0001

with p-values (with both FF_{AFL} and FF_{AFL++} as baseline) in https://github.com/HexHive/FishFuzz.

A.3 Hyperparameter Study

FishFuzz's performance depends on a set of inter-dependent parameters.

Different Exploration/Exploitation Timeout. We measure coverage, the number of reached/triggered targets on our UBSan benchmark with FF_{AFL++}. We leverage different timeouts for *inter-function* (15, 20, 30 mins), *intra-function*(10, 20 mins), and *exploitation* phases (30, 40, 60 mins). The performance changes as follows comparing the best/worst settings: in `tcpdump`, the coverage improves by 12.03%, while in `readelf` the reached targets improve by 15.43%. We also notice that in `tiff2ps` the number of triggered targets improves by 37.29%.

Target Ranking Threshold. We also study the impact of different target ranking thresholds (*i.e.,* the variable *threshold* in line 14). We evaluate our UBSan benchmark for 5 rounds with threshold top 10%, 20%, and 40% fewer hit targets, respectively. The results suggest the threshold setting only affects 0.60% of the coverage and 2.91% of the triggered targets respectively. Therefore, this hyperparameter does not affect the capability of FishFuzz to exploit/reach targets.

A.4 Ablation Study

We run a campaign on our UBSan dataset (24h each for 5 rounds) by using only *inter-function* and *exploitation* (§4.2). This experiment shows the independent contribution for edge/-function coverage and reached/triggered targets. Note that *intra-function* (Figure 4) is the original AFL/AFL++ one, thus we ignore it. As shown in Table 13, the combination of all phases produces the best results. FF_{AFL} covers 5.29% more edges and triggered 16.02% compared against best of *inter-function* and *exploitation* stage. For the *inter-function*, although it reaches 3.73% less basic block compared with *exploitation* stage, it still finds slightly more functions than *exploitation* (1.54%).

A.5 Time-To-Exposure Details

Table 14 and Table 15 show the unique bugs found in ASan and the Two-Stage benchmarks (more info in §6 "Benchmarks Selected") and the respective time-to-exposure.

Table 12: p-values of the Mann-Whitney U test from the experiments inTable 5.

program	FF_{AFL}		FF_{AFL++}		AFL		AFL++		AFLFast		FairFuzz		EcoFuzz		KScheduler	
	cov	reach	cov	reach	cov	reach	cov	reach	cov	reach	cov	reach	cov	reach	cov	reach
cflow	0.0122	0.0381	1	1	0.0026	0.0001	0.0002	0.0001	0.0243	0.0001	0.0026	0.0001	0.0002	0.0004	0.0002	0.0001
mujs	0.0073	0.0036	1	1	0.0002	0.0002	0.0008	0.0003	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002
tic	0.0451	0.0407	1	1	0.0003	0.0002	0.0017	0.0013	0.0002	0.0002	0.0004	0.0003	0.0013	0.0013	0.0003	0.0002
mupdf	0.0101	0.0957	1	1	0.0002	0.0003	0.1403	0.0113	0.0065	0.001	0.1212	0.0112	0.0091	0.001	0.0001	0.0001
dwarfdump	0.0139	0.4457	1	1	0.1402	0.8197	0.6219	0.3416	0.0003	0.0003	0.9096	0.0807	0.2411	0.9696	0.2727	0.1846
w3m	0.289	0.676	1	1	0.0001	0.0001	0.0019	0.0002	0.0001	0.0001	0.0001	0.0001	0.0125	0.068	0.0002	0.0002
cxxfilt	0.0002	0.0002	1	1	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002

Table 13: average edge/function coverage and target reached/triggered in UBSan benchmark. all means combination of all strategies (FF_{AFL}), explore is Algorithm 1 only and exploit is Algorithm 2 only.

programs	all				explore				exploit			
	cov	func	reach	trigger	cov	func	reach	trigger	cov	func	reach	trigger
djpeg	12427	147.2	3969.4	140.4	11621.8	141	3842	133.8	11651.6	141	3846.8	134
jasper	10775.2	468.2	1772.6	50.4	9783.2	449	1609.6	33.2	9592.4	436.2	1587.2	31.2
readelf	2008	51.2	157.6	28	2112	53.2	178	27.6	1889.4	49.4	149.2	19.6
objdump	10320.4	292.6	1118.8	97.6	9004.6	271.8	985.2	77.4	9154.6	268.2	982.6	87.6
tcpdump	24422.8	471.8	3822.6	154.8	22972.8	461.4	3596.6	146	24279	439.6	3678	131.2
tiff2pdf	13076.8	282.4	1919	19.4	11104	270.6	1424.6	12.4	12673.8	275.2	1847.4	15.8
tiff2ps	9088	198.4	1177.8	15.8	8457.6	189.8	1070.2	8.6	8738.6	191.4	1129.2	10.6
xmllint	8069.4	306.8	723.6	15	7523.8	298.2	679.4	10.4	7681	301.6	690	11.6
sum	90187.6	2218.6	14661.4	521.4	82579.8	2135	13385.6	449.4	85660.4	2102.6	13910.4	441.6

Table 14: Time-To-Exposure of 30 real world bugs found after 10 rounds of 24 hours in the Two Stage benchmark.

BUG	programs	stack trace	type	FF_{AFL}	FF_{AFL++}	AFL	AFLFast	FairFuzz	EcoFuzz	K-Scheduler	AFL++
CVE-2019-16165	cflow	reference->expression	UAF	2:27:54	**0:19:32**	10:18:24	9:02:05	10:10:57	9:52:40	1:54:52	0:24:49
CVE-2019-16166	cflow	nexttoken->parse_function_declaration	HOF	15:54:06	**9:36:24**	19:30:12	20:11:16	17:19:06	20:45:32	13:06:41	14:01:42
cflow_unknow_1	cflow	collect_processor->hash_do_for_each->collect_symbols	HOF	18:51:31	**16:06:51**	24:00:00	22:51:20	20:47:46	13:25:35	21:19:50	24:00:00
CVE-2022-30974	mujs	compile->compile	stack exhastion	**0:59:15**	14:56:30	24:00:00	24:00:00	24:00:00	24:00:00	24:00:00	24:00:00
mujs_unknow_1	mujs	jsP_foldconst->jsP_foldconst	stack exhastion	**1:53:49**	15:48:42	24:00:00	24:00:00	24:00:00	24:00:00	24:00:00	24:00:00
mujs_unknow_2	mujs	unary->unary	stack exhastion	**1:01:13**	16:19:47	24:00:00	24:00:00	24:00:00	11:55:31	24:00:00	24:00:00
mujs_unknow_3	mujs	count->count	stack exhastion	**20:24:15**	22:40:38	24:00:00	24:00:00	24:00:00	24:00:00	24:00:00	24:00:00
CVE-2018-5759	mujs	jsC_cexp->jsC_cexp	stack exhastion	**13:30:49**	21:46:19	24:00:00	24:00:00	24:00:00	24:00:00	24:00:00	24:00:00
CVE-2017-5627	mujs	js_pushstring->Fp_toString	HOF	14:21:28	10:40:18	17:54:54	12:30:47	15:24:11	19:51:40	17:00:15	**8:16:52**
CVE-2018-6191	mujs	js_strtod->lexnumber	GOF	23:57:36	19:54:45	21:38:13	24:00:00	23:23:10	22:11:11	24:00:00	**19:26:52**
tic_leak_1	tic	strdup->_nc_set_writedir->main	leak	**13:20:26**	20:36:24	24:00:00	24:00:00	24:00:00	24:00:00	24:00:00	24:00:00
tic_leak_2	tic	malloc->_nc_init_termtype->_nc_init_entry	leak	**9:27:45**	10:23:48	24:00:00	21:52:12	20:56:50	20:03:57	24:00:00	15:30:04
tic_leak_3	tic	malloc->_nc_resolve_uses2->main	leak	**16:39:44**	22:49:26	24:00:00	24:00:00	24:00:00	24:00:00	24:00:00	22:24:48
CVE-2021-39537	tic	_nc_captoinfo->_nc_parse_entry->_nc_read_entry_source	HOF	17:36:37	**1:01:29**	4:32:25	11:47:47	7:06:39	15:02:31	21:55:41	3:54:42
CVE-2017-13731	tic	postprocess_termcap->_nc_parse_entry->_nc_read_entry_source	HOF	24:00:00	22:52:34	24:00:00	24:00:00	24:00:00	24:00:00	24:00:00	**21:38:29**
CVE-2022-29458	tic	convert_strings->_nc_read_termtype	HOF	17:23:15	**10:39:59**	24:00:00	24:00:00	24:00:00	24:00:00	24:00:00	24:00:00
CVE-2017-13729	tic	__interceptor_strlen.part.36->_nc_save_str	UAF	20:10:17	**19:11:37**	24:00:00	24:00:00	24:00:00	24:00:00	24:00:00	24:00:00
CVE-2017-17858	mutool	ensure_solid_xref->pdf_get_xref_entry	HOF	9:59:33	18:56:29	17:47:20	17:17:42	**5:48:57**	19:19:19	24:00:00	11:28:14
CVE-2016-6265	mutool	pdf_load_xref->pdf_init_document	OBW	24:00:00	24:00:00	21:32:51	24:00:00	**15:57:10**	19:49:54	24:00:00	19:33:02
CVE-2019-14249	dwarfdump	read_gs_section_group	FPE	0:00:05	0:00:02	0:00:03	0:23:27	0:23:43	0:24:23	0:23:19	0:00:02
CVE-2022-39170	dwarfdump	_dwarf_destruct_elf_nlaccess	double free	4:21:22	10:25:58	**1:20:58**	12:45:42	2:41:26	6:44:44	5:25:28	10:13:38
dwarf_unknow_1	dwarfdump	dwarf_pe_load_dwarf_section_headers	HOF	**23:50:30**	24:00:00	24:00:00	24:00:00	24:00:00	24:00:00	24:00:00	24:00:00
CVE-2018-18484	cxxfilt	cplus_demangle_type->d_bare_function_type->d_function_type	stack exhastion	**1:08:31**	24:00:00	16:37:15	22:34:27	24:00:00	7:35:09	20:34:34	24:00:00
CVE-2018-17985	cxxfilt	cplus_demangle_type->cplus_demangle_type	stack exhastion	**1:12:08**	24:00:00	13:33:11	22:33:52	24:00:00	1:35:54	18:01:46	24:00:00
cxxfilt_unknow_1	cxxfilt	d_pointer_to_member_type->cplus_demangle_type	stack exhastion	**3:03:39**	24:00:00	19:40:40	22:34:38	24:00:00	7:05:33	23:17:13	24:00:00
CVE-2018-18700	cxxfilt	d_name->d_encoding->d_local_name	stack exhastion	**1:11:57**	24:00:00	19:21:04	22:32:31	24:00:00	4:58:06	19:53:30	24:00:00
CVE-2018-9138	cxxfilt	demangle_nested_args->do_type->do_arg->demangle_args	stack exhastion	**2:01:55**	23:13:48	21:33:11	24:00:00	24:00:00	4:09:24	21:51:43	21:37:46
cxxfilt_unknow_2	cxxfilt	d_template_arg->d_template_args_1	stack exhastion	**5:24:50**	24:00:00	24:00:00	24:00:00	24:00:00	16:52:02	20:05:49	24:00:00
CVE-2019-9071	cxxfilt	d_count_templates_scopes->d_count_templates_scopes	stack exhastion	**6:04:27**	24:00:00	24:00:00	24:00:00	24:00:00	24:00:00	24:00:00	24:00:00
CVE-2019-9070	cxxfilt	d_expression_1->d_expression_1	stack exhastion	**4:48:47**	24:00:00	24:00:00	24:00:00	24:00:00	21:39:56	24:00:00	24:00:00
CVE-2018-9996	cxxfilt	demangle_real_value->demangle_template_value_parm	stack exhastion	20:17:37	24:00:00	23:27:07	24:00:00	24:00:00	**20:13:10**	24:00:00	24:00:00

Table 15: Time-To-Exposure of 72 real world bugs found after 10 rounds of 60 hours in the ASan benchmark.

bug id	programs	stack trace	bug type	FF_{AFL}	FF_{AFL++}	AFL	AFL++	ParmeSan	TortoiseFuzz
CVE-2018-19325	tcpdump	EXTRACT_32BITS->mfr_print	heap overflow	38:26:44	**35:53:56**	50:48:10	56:36:43	60:00:00	60:00:00
CVE-2017-16808	tcpdump	lookup_emem->etheraddr_string	heap overflow	60:00:00	**57:19:30**	59:48:41	60:00:00	60:00:00	60:00:00
catdoc_issue_8	catdoc	find_file	null pointer	**13:36:47**	60:00:00	9:58:12	60:00:00	60:00:00	60:00:00
CVE-2020-18898	exiv2	printIFDStructure->printIFDStructure	stack exhaustion	**15:46:29**	40:10:20	60:00:00	31:50:12	60:00:00	60:00:00
CVE-2018-17282	exiv2	Exiv2::DataValue::copy	SEGV	**48:25:23**	50:31:59	60:00:00	54:37:08	60:00:00	60:00:00
flvmeta_issue_13	flvmeta	__interceptor_strcmp.part.26 ->xml_on_metadata_tag_only	SEGV	0:00:00	0:00:00	0:00:00	0:00:00	0:05:17	0:00:00
flvmeta_issue_14	flvmeta	__strftime_l ->xml_amf_data_dump	SEGV	0:13:34	0:07:40	0:07:26	**0:04:04**	20:32:55	0:18:13
liblouis_issue_728-1	lou_checktable	vsnprintf->compileError	stack-overflow	**20:05:21**	29:34:34	50:44:01	60:00:00	60:00:00	59:40:42
liblouis-java_issue_10	lou_checktable	compileUplow ->compileRule	UAF	**36:39:59**	38:28:58	60:00:00	60:00:00	60:00:00	60:00:00
liblouis_issue_573	lou_checktable	pattern_compile_expression->pattern_compile_expression	Stack Exhaustion	**53:31:54**	53:52:37	60:00:00	60:00:00	60:00:00	60:00:00
CVE-2018-11685	lou_checktable	compileHyphenation->compileRule	stack overflow	60:00:00	**40:19:42**	60:00:00	60:00:00	60:00:00	60:00:00
CVE-2018-8883	nasm	parse_line ->assemble_file	global-buffer-overflow	0:03:04	0:02:46	**0:02:02**	0:02:21	60:00:00	0:05:51
CVE-2019-6291	nasm	expr6 ->expr6	stack-overflow	**23:58:18**	60:00:00	60:00:00	60:00:00	60:00:00	60:00:00
CVE-2018-8882	nasm	ieee_shr ->to_float	stack-underflow	16:29:29	**6:24:29**	47:39:19	25:51:22	60:00:00	60:00:00
CVE-2021-33457	nasm	expand_mmac_params->pp_getline	heap overflow	60:00:00	**57:34:31**	60:00:00	60:00:00	60:00:00	60:00:00
CVE-2018-16517	nasm	islocal ->find_label	SEGV	48:33:10	**28:15:12**	51:25:28	32:44:35	60:00:00	59:56:40
CVE-2018-18484	nm-new	cplus_demangle_type ->d_bare_function_type ->d_function_type	Stack Exhaustion	**28:44:07**	42:01:53	48:18:18	60:00:00	60:00:00	59:15:02
CVE-2018-9138	nm-new	do_type->do_arg	Stack Exhaustion	**24:43:11**	42:30:07	46:57:50	59:49:47	60:00:00	51:02:03
CVE-2019-9070	nm-new	d_expression_1 ->d_expression_1	Stack Exhaustion	**32:00:16**	41:54:22	60:00:00	60:00:00	60:00:00	60:00:00
CVE-2018-17985	nm-new	cplus_demangle_type ->cplus_demangle_type	Stack Exhaustion	**20:18:26**	39:15:26	48:14:18	60:00:00	60:00:00	58:44:38
binutils-unknow-1	nm-new	cplus_demangle_type ->d_pointer_to_member_type	Stack Exhaustion	**12:00:38**	54:34:15	53:07:52	60:00:00	60:00:00	60:00:00
CVE-2019-9071	nm-new	d_count_templates_scopes ->d_count_templates_scopes	Stack Exhaustion	**25:32:00**	41:35:26	60:00:00	60:00:00	60:00:00	60:00:00
CVE-2018-18700	nm-new	d_local_name ->d_name ->d_encoding	Stack Exhaustion	**20:05:55**	39:14:25	50:40:09	60:00:00	60:00:00	59:05:22
binutils-unknow-2	nm-new	d_template_args_1 ->d_template_args ->d_template_arg	Stack Exhaustion	**04:02:40**	49:18:08	60:00:00	60:00:00	60:00:00	60:00:00
CVE-2018-9996	nm-new	demangle_real_value->demangle_template_value_parm	Stack Exhaustion	**48:34:55**	58:00:44	60:00:00	60:00:00	60:00:00	60:00:00
binutils_issue_1375935	nm-new	d_print_comp->d_print_comp_inner	Stack Exhaustion	58:50:22	**58:37:44**	60:00:00	60:00:00	60:00:00	60:00:00
CVE-2018-20552	tcpprep	packet2tree ->add_tree_ipv4	heap overflow	14:40:00	0:04:11	**0:01:09**	12:03:27	60:00:00	0:10:09
CVE-2020-24266	tcpprep	get_l2len ->get_ipv4	heap overflow	1:27:17	1:35:06	**0:13:18**	0:52:37	60:00:00	1:14:46
CVE-2018-17795	tiff2pdf	t2p_write_pdf->main	heap overflow	**58:13:04**	60:00:00	60:00:00	60:00:00	60:00:00	60:00:00
libtiff_issue_258	tiff2pdf	t2p_read_tiff_data->t2p_write_pdf	null pointer	57:22:05	59:43:58	60:00:00	54:19:59	60:00:00	**53:09:29**
ngiflib_issue_11	gif2tga	WritePixel ->DecodeGifImg	heap overflow	0:00:00	0:00:00	0:00:00	0:00:00	0:00:00	0:00:00
ngiflib_issue_1	gif2tga	WritePixels ->DecodeGifImg	heap overflow	0:00:04	0:00:03	0:00:04	0:00:02	0:00:14	0:00:06
CVE-2019-20219	gif2tga	GifIndexToTrueColor ->WritePixels	heap overflow	0:00:04	0:00:03	0:00:04	0:00:02	0:06:38	0:00:07
ngiflib_issue_4	gif2tga	DecodeGifImg ->LoadGif	heap overflow	0:00:04	0:00:03	0:00:04	0:00:04	0:08:35	0:00:07
gpac_issue_1333	MP4Box	gf_isom_box_dump_ex->gf_isom_box_array_dump	UAF	0:00:40	0:00:28	0:00:40	**0:00:18**	18:42:19	0:02:21
gpac_issue_1661	MP4Box	gf_isom_box_del->gf_isom_box_del	UAF	0:02:37	0:01:34	0:02:19	**0:01:04**	0:02:54	0:08:15
gpac_issue_1099	MP4Box	gf_isom_oinf_read_entry->gf_isom_box_parse_ex	heap overflow	0:38:38	2:05:03	1:44:01	1:51:07	**0:33:06**	2:07:35
gpac_issue_1250	MP4Box	gf_m2ts_process_data->gf_import_mpeg_ts	heap overflow	**15:17:10**	29:21:37	23:39:24	56:02:08	60:00:00	36:55:54
gpac_issue_1422	MP4Box	gf_m2ts_section_complete->gf_m2ts_gather_section	heap overflow	**3:49:32**	29:51:36	37:24:53	59:02:10	60:00:00	58:46:16
gpac_issue_1446	MP4Box	gf_m2ts_get_adaptation_field->gf_m2ts_process_data	stack overflow	**3:12:32**	27:46:30	39:13:11	57:48:21	60:00:00	60:00:00
CVE-2018-13005	MP4Box	urn_Read->gf_isom_box_parse_ex	heap overflow	**18:32:11**	22:18:47	15:57:09	51:38:52	60:00:00	33:33:56
gpac_unknow_1	MP4Box	gf_isom_sample_entry_get_bitrate->AVC_RewriteESDescriptorEx	UAF	12:19:05	33:58:37	49:39:20	58:38:01	60:00:00	**26:23:00**
CVE-2018-7752	MP4Box	gf_media_avc_read_sps->avcc_Read	stack overflow	19:13:32	5:58:45	25:31:18	**4:14:43**	20:01:51	31:10:59
CVE-2018-21015	MP4Box	AVC_DuplicateConfig->merge_avc_config	null pointer	51:20:32	53:43:02	57:46:19	60:00:00	**48:39:58**	60:00:00
CVE-2019-20169	MP4Box	trak_Read->gf_isom_box_parse_ex	UAF	16:24:41	27:59:42	18:36:35	**2:08:12**	2:11:47	41:07:32
gpac_issue_1096	MP4Box	metx_Read->gf_isom_box_parse_ex	heap overflow	**40:03:46**	49:09:00	46:08:35	48:12:03	60:00:00	51:47:14
gpac_issue_1097	MP4Box	printf_common->ReportGenericError	heap overflow	**22:12:04**	60:00:00	60:00:00	51:19:56	60:00:00	47:19:37
CVE-2019-20630	MP4Box	BS_ReadByte->gf_bs_read_int	heap overflow	**34:10:31**	52:56:22	58:25:29	60:00:00	60:00:00	60:00:00
gpac_unknow_2	MP4Box	gf_m2ts_process_tdt_tot->gf_m2ts_section_complete	heap overflow	**22:07:45**	40:21:03	46:47:03	60:00:00	60:00:00	60:00:00
CVE-2019-20629	MP4Box	gf_m2ts_process_pmt->gf_m2ts_section_complete	heap overflow	**48:20:55**	52:10:17	55:44:50	60:00:00	60:00:00	60:00:00
CVE-2019-20632	MP4Box	gf_odf_delete_descriptor	null pointer	**50:10:54**	57:45:29	59:20:16	60:00:00	60:00:00	60:00:00
gpac_issue_1445	MP4Box	__asan_memcpy->gf_bs_read_data	heap overflow	48:55:11	55:55:11	59:00:47	60:00:00	**5:03:02**	60:00:00
gpac_issue_1317	MP4Box	GF_IPMPX_ReadData->gf_ipmpx_data_parse	heap overflow	55:22:35	60:00:00	60:00:00	60:00:00	60:00:00	60:00:00
gpac_unknow_3	MP4Box	gf_list_enum->gf_odf_delete_descriptor_list	heap overflow	50:24:04	57:44:56	59:20:34	60:00:00	60:00:00	60:00:00
gpac_issue_1327-1	MP4Box	gf_isom_box_parse_ex->gf_isom_box_array_read_ex	heap overflow	60:00:00	60:00:00	60:00:00	60:00:00	**0:26:17**	60:00:00
gpac_unknow_4	MP4Box	gf_isom_parse_movie_boxes	null pointer	60:00:00	57:25:18	**38:59:37**	54:17:28	60:00:00	60:00:00
gpac_unknow_5	MP4Box	gf_m2ts_process_sdt->gf_m2ts_section_complete	heap overflow	**46:00:58**	52:43:55	52:40:11	60:00:00	60:00:00	60:00:00
gpac_issue_1332	MP4Box	stbl_AddBox->gf_isom_box_array_read_ex	null pointer	60:00:00	60:00:00	60:00:00	60:00:00	**36:53:28**	60:00:00
gpac_unknow_6	MP4Box	sgpd_dump->gf_isom_box_dump_ex	heap overflow	60:00:00	60:00:00	60:00:00	60:00:00	**45:56:20**	60:00:00
gpac_unknow_7	MP4Box	abst_Read->gf_isom_box_parse_ex	heap overflow	**52:18:32**	60:00:00	56:48:01	60:00:00	58:56:05	60:00:00
gpac_issue_1446	MP4Box	on_m2ts_import_data->gf_m2ts_reframe_nalu_video	heap overflow	**58:45:40**	60:00:00	60:00:00	60:00:00	60:00:00	60:00:00
gpac_unknow_8	MP4Box	gf_m2ts_reframe_aac_adts->gf_m2ts_flush_pes	heap overflow	**59:12:06**	60:00:00	60:00:00	60:00:00	60:00:00	60:00:00
gpac_unknow_9	MP4Box	gf_mp3_get_next_header_mem->gf_m2ts_reframe_mpeg_audio	heap overflow	**59:14:31**	60:00:00	60:00:00	60:00:00	60:00:00	60:00:00
gpac_unknow_10	MP4Box	__asan_memcpy->gf_m2ts_reframe_aac_latm	heap overflow	**59:15:20**	60:00:00	60:00:00	60:00:00	60:00:00	60:00:00
gpac_unknow_11	MP4Box	__interceptor_memchr.part.36->gf_m2ts_reframe_nalu_video	heap overflow	**59:16:03**	60:00:00	60:00:00	60:00:00	60:00:00	60:00:00
gpac_unknow_12	MP4Box	gf_m2ts_reframe_mpeg_video->gf_m2ts_flush_pes	heap overflow	**59:24:00**	60:00:00	60:00:00	60:00:00	60:00:00	60:00:00
gpac_unknow_13	MP4Box	audio_sample_entry_AddBox->gf_isom_box_array_read_ex	null pointer	60:00:00	60:00:00	60:00:00	60:00:00	**55:19:00**	60:00:00
CVE-2018-13006	MP4Box	__interceptor_strlen.part.24->hdlr_dump	heap overflow	55:50:01	**23:23:05**	60:00:00	58:13:36	60:00:00	60:00:00
gpac_issue_1348	MP4Box	dimC_Read->gf_isom_box_parse_ex	heap overflow	**57:38:37**	60:00:00	60:00:00	60:00:00	60:00:00	60:00:00
gpac_issue_1262	MP4Box	txtc_Read->gf_isom_box_parse_ex	heap overflow	**59:21:35**	60:00:00	60:00:00	60:00:00	60:00:00	60:00:00
gpac_issue_1328	MP4Box	GF_IPMPX_AUTH_Delete->gf_ipmpx_data_del	heap overflow	**57:03:56**	60:00:00	60:00:00	60:00:00	60:00:00	60:00:00
gpac_issue_1338	MP4Box	ilst_item_Read->gf_isom_box_parse_ex	null pointer	58:11:06	**47:49:35**	49:06:22	60:00:00	60:00:00	59:52:36
gpac_unknow_14	MP4Box	ilst_item_del->gf_isom_box_del	heap overflow	60:00:00	**48:50:21**	49:42:13	60:00:00	60:00:00	60:00:00