BeDivFuzz: Integrating Behavioral Diversity into Generator-based Fuzzing

Hoang Lam Nguyen
Humboldt-Universität zu Berlin
Germany
nguyehoa@informatik.hu-berlin.de

Lars Grunske
Humboldt-Universität zu Berlin
Germany
grunske@informatik.hu-berlin.de

ABSTRACT

A popular metric to evaluate the performance of fuzzers is branch coverage. However, we argue that focusing solely on covering many different branches (i.e., the *richness*) is not sufficient since the majority of the covered branches may have been exercised only once, which does not inspire a high confidence in the reliability of the covered code. Instead, the distribution of the executed branches (i.e., the *evenness*) should also be considered. That is, *behavioral diversity* is only given if the generated inputs not only trigger many different branches, but also trigger them evenly often with diverse inputs. We introduce BeDivFuzz, a feedback-driven fuzzing technique for generator-based fuzzers. BeDivFuzz distinguishes between *structure-preserving* and *structure-changing* mutations in the space of syntactically valid inputs, and biases its mutation strategy towards validity and behavioral diversity based on the received program feedback. We have evaluated BeDivFuzz on Ant, Maven, Rhino, Closure, Nashorn, and Tomcat. The results show that BeDivFuzz achieves better behavioral diversity than the state of the art, measured by established biodiversity metrics, namely the Hill numbers, from the field of ecology.

KEYWORDS

Structure-aware fuzzing, behavioral diversity, random testing

ACM Reference Format:
Hoang Lam Nguyen and Lars Grunske. 2022. BeDivFuzz: Integrating Behavioral Diversity into Generator-based Fuzzing. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510182

1 INTRODUCTION

Traditionally, fuzzing tools (e.g., [9, 26, 33, 34, 43, 47, 53]) have been used to evaluate the software under test (SUT) with respect to security and robustness properties. Typically, vulnerabilities are found by feeding the SUT malformed inputs, potentially resulting in unexpected program behavior, which can be identified using e.g., memory and safety oracles [29, 37, 41]. Since most of the vulnerabilities emerge due to incorrect handling of unexpected inputs, security-oriented fuzzers usually target the input parsing and processing stages of the SUT. Recently, there is a trend [3, 32, 35, 45, 51, 54] to use fuzzers to test the actual core functionality of the SUT, rather than the early input processing stages only. The challenge of testing the core functionality of a program that expects complex structured inputs is mainly due to the following problems:

(1) The input must be *syntactically* valid (i.e., conform to the expected structure/type) in order to be parseable by the SUT.
(2) The input must satisfy any additional *semantic* validity constraints (e.g., assertions or `repOk` methods [27]) to actually reach the core program functionality.
(3) The generated inputs must exhibit some sort of *diversity* in order to trigger diverse behavior.

The techniques targeting this problem typically rely on an input specification (e.g., a grammar) that describes the expected input structure to produce inputs of the expected format. Generator-based fuzzers like Zest [32] follow an imperative approach, where the tester implements a *generator* program that is able to produce syntactically valid inputs. Zest uses code coverage and validity feedback to search for inputs that exercise many different branches in the semantic analysis stages of the SUT. That is, the goal of Zest is to *cover* as much of the semantic program behavior as possible. A more recent technique, RLCheck [35], utilizes reinforcement learning to automatically guide the generator towards high input diversity, i.e., inputs that exercise different traces. However, while RLCheck is able to produce a large number of diverse inputs that trigger *specific behaviors*, it fails to cover many different behaviors.

We are interested in a technique that is able to not only *cover* many different behaviors, but also test these behaviors in a *diverse* manner. That is, we aim for an even distribution of exercised behaviors, rather than focusing our testing effort solely on particular behaviors. A simplified illustration of this idea can be found in Figure 1. In the figure, the grey areas in the input space represent the valid inputs (i.e., inputs that trigger the core functionality of the SUT), whereas the white areas represent other invalid inputs (e.g., inputs that only trigger error handling code). Traditional fuzzers like AFL [53], as depicted in Figure 1.a, tend to produce malformed inputs and hence only cover a small proportion of the valid behavior. Further, the distribution of the triggered valid behavior mostly concentrates on the more likely branches. On the other hand, validity-focused fuzzers like Zest and RLCheck (cf. Figure 1.b) either only focus on covering many different behaviors (but not their diverse execution, indicated by grey areas with a low concentration of green crosses) or disproportionally test particular behaviors only (indicated by the small areas with a high number of green crosses). Our proposed technique, BeDivFuzz (cf. Figure 1.c), avoids this problem by searching for many different valid inputs while evenly testing the behaviors in a diverse manner.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510182

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:10 UTC from IEEE Xplore. Restrictions apply.

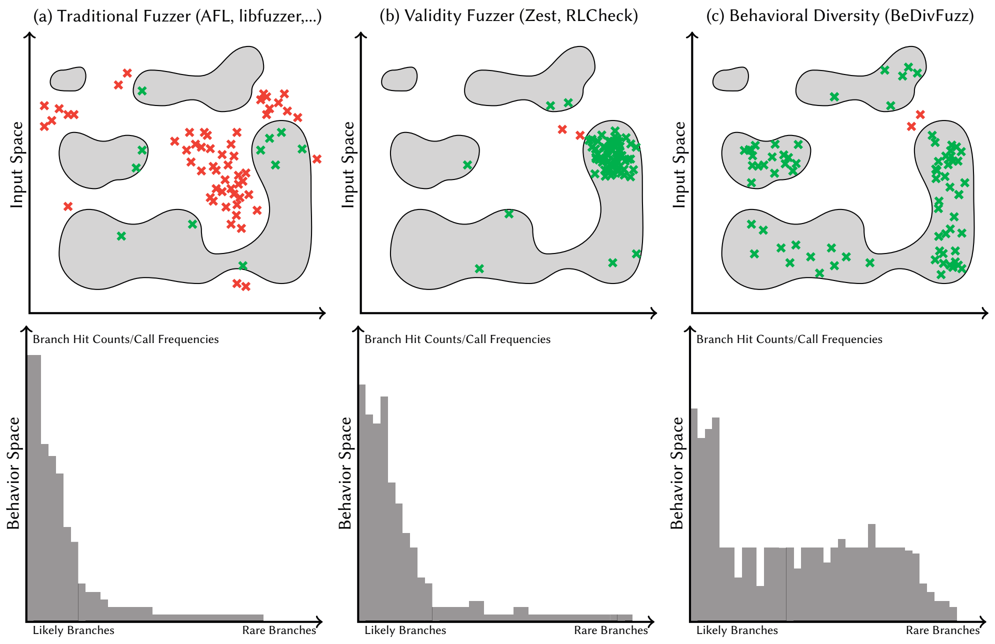


Figure 1: Illustrative simplified motivation for BeDivFuzz. In the input space, the white and grey areas represent invalid and valid input areas for the SUT, respectively. The green crosses represent valid test inputs, while the red crosses represent invalid inputs. In line with [39], we use branch hit counts and call frequencies as an illustrative metric to represent the behavior space.

In particular, BeDivFuzz distinguishes between *structure-changing* and *structure-preserving* mutations. Performing structure-changing mutations allows to search for new program behavior that is only triggered if the input satisfies particular structural properties. In contrast, the structure-preserving mutations allow to target specific behavior of the code with different variants of the same input structure, effectively testing the targeted behavior in a diverse manner. BeDivFuzz uses an adaptive mutation strategy that utilizes the received program feedback to guide the search towards high behavioral diversity. In order to determine the behavioral diversity of a fuzzing campaign, we propose a novel metric that incorporates both the number of covered branches and the branch execution distribution over all unique traces.

Novelty and Contributions. Overall, we provide the following key contributions:

- We present BeDivFuzz, a novel fuzzing approach that generates valid and behaviorally diverse inputs.
- We propose to utilize Hill numbers [17], a common metric for species diversity from ecology, to quantify the behavioral diversity of the covered branches by a fuzzing campaign.
- We evaluate BeDivFuzz based on several fuzzing campaigns with XML and JavaScript SUTs, namely Ant, Maven, Closure, Rhino, Nashorn, and Tomcat.
- We provide the source code of BeDivFuzz and a replication package of our results at https://github.com/hub-se/BeDivFuzz.

Significance of the Contributions. We provide a novel metric that quantifies the *behavioral diversity* of a fuzzing campaign and propose a technique that improves upon the current state-of-the-art w.r.t. to that metric. The proposed metric shifts the focus of simply *covering many behaviors* (i.e., branch coverage) to *diversely testing many different behaviors.* Thus, it will potentially serve as a stepping stone towards future systematic evaluations of a SUT with respect to reliability and correctness after the termination of a fuzzing campaign [6, 7], as desired in practical software engineering.

```
def generate_tree(depth=0):
  value = random.randint(0, 10)
  tree = BinaryTree(value)
  if depth >= MAX_DEPTH:
    return tree
  if random.choose([True, False]):
    tree.left = generate_tree(depth+1)
  if random.choose([True, False]):
    tree.right = generate_tree(depth+1)
  return tree
```

Figure 2: A simple binary tree generator. Adapted from [35].

2 BACKGROUND

2.1 Generator-Based Fuzzing

Fuzzing (also known as fuzz testing) attempts to find bugs and crashes in software through random input generation. A common challenge in fuzzing is to produce test inputs for programs that expect complex structured inputs (e.g., compilers). Effectively testing these types of programs is challenging due to the following reasons:

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:10 UTC from IEEE Xplore. Restrictions apply.

(i) The generated inputs must be *syntactically* valid in order to be successfully parsed by the program. (ii) The input must also satisfy any additional *semantic* validity constraints in order to actually exercise the SUT's core functionality. (iii) The test inputs have to be *diverse* enough to execute a variety of different program behavior. Structure-aware fuzzers typically approach this problem by utilizing domain-specific knowledge about the expected input type or format. One way to provide this knowledge to the fuzzer is via a declarative input specification (e.g., a grammar [2, 14, 16, 18, 30, 39, 48]). Alternatively, the tester may write an imperative *generator* program that randomly samples syntactically valid inputs, an approach known as *generator-based fuzzing* [19, 31, 32]. For example, Figure 2 shows the pseudocode for a generator that produces random binary trees, which will serve as the running example throughout the paper. The function `generate_tree` first samples a random integer value between 0 and 10 (Line 2) to instantiate the root node (Line 3). If a user-defined maximum depth has been reached, the generated node is returned (Lines 4–5). Otherwise, the generator non-deterministically decides whether to generate left and right child nodes, in which case `generate_tree` is recursively called (Lines 6–9). The output of the generator thus depends on the sequence of *random choices* made during the generation process. In the example, the sequence of random choices precisely determines the shape of the binary tree and all the tree node values. As a concrete example, consider the following random choice sequence for the call of `generate_tree` (`MAX_DEPTH = 5`):

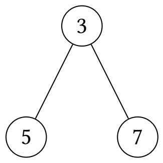
Random choice → result	**Context**
`random.randint` → `3`	Root: node value (Line 2)
`random.choose` → `False`	Root: generate left child? (Line 6)
`random.choose` → `False`	Root: generate right child? (Line 8)

This sequence of random choices would result in a "binary-tree" consisting of a single root node with the value 3 and no child nodes:

To give a different example, consider the following choice sequence:

Random choice → result	**Context**
`random.randint` → `3`	Root: node value (Line 2)
`random.choose` → `True`	Root: generate left child? (Line 6)
`random.randint` → `5`	Left child: node value (Line 2)
`random.choose` → `False`	Left child: gen. left child? (Line 6)
`random.choose` → `False`	Left child: gen. right child? (Line 8)
`random.choose` → `True`	Root: generate right child? (Line 8)
`random.randint` → `7`	Right child: node value (Line 2)
`random.choose` → `False`	Right child: gen. left child? (Line 6)
`random.choose` → `False`	Right child: gen. right child? (Line 8)

The corresponding binary tree consists of a root node with the value 3 and leaf nodes with the values 5 and 7 as the left and right child, respectively:



A typical challenge in generator-based testing arises when the input is expected to satisfy complex semantic validity constraints that are not explicitly considered by the generator. To continue the current example, the SUT may expect the generated tree to be sorted (e.g., a binary search tree) or height-balanced (e.g., an AVL tree). In this case, random input sampling is usually not very effective as most of the generated inputs will be rejected by the SUT. Instead, recent techniques bias the generator towards producing valid inputs by directly controlling the random choices made during the input generation process.

Zest [32] relies on the fact that the implementation of random number generators usually depends on a pseudo-random source of *untyped bits*, which Padhye et al. refer to as *parameters*. Specifically, random values of a particular type are generated by consuming and interpreting a fixed number of parameters. For instance, a random boolean value may be sampled by consuming one bit from the source of untyped parameters, which is then interpreted as `False` if it is zero and otherwise as `True`. The pseudo-random nature of this method ensures that using the same random seed always generates the same sequence of parameters. For a given generator, this will produce the same sequence of random choices, effectively resulting in the same generated input. On the other hand, by mutating the untyped parameter sequence, the sequence of random choices can be altered, which corresponds to complex structural mutations in the input domain. For example, mutating the parameters that are consumed by the first call to `random.randint()` (Figure 2, Line 2) results in the value of the root node to be mutated. Similarly, mutating the parameters corresponding to a call to `random.choose()` (Line 6) may change the decision on whether a left child node will be generated. As a result, by controlling the sequence of parameters, it is possible to directly control the output of the input generator. Zest [32] exploits this insight by performing a feedback-directed parameter search to guide the generators towards high coverage in the semantic analysis stages. Contrary, RLCheck [35] employs reinforcement learning to automatically learn a *guide* that leads the generator to produce diverse valid inputs (w.r.t. a validity function).

2.2 Hill-Numbers

In the field of ecology, researchers are interested in quantifying the biodiversity of an assemblage based on a sample of species. A commonly used class of diversity metrics are the Hill numbers [17]. The Hill numbers consider both species richness (i.e., the total number of different species) and species abundances (i.e., the number of individuals per species) in a sample, and are defined as follows:

Definition 2.1 (Hill number of order q)**.** Let S be the species richness and p_i the relative abundance of the i-th species in the dataset. The *Hill number of order q* is defined as (${}^qD$ is the original notation):

$$ {}^qD = D(q) = \left(\sum_{i=1}^{S} p_i^q \right)^{1/(1-q)} \tag{1} $$

For $q = 1$, Equation 1 is undefined, but its limit for $q \to 1$ corresponds to the exponential of the Shannon(-Wiener) index [38, 40]:

$$ {}^1D = D(1) = \lim_{q \to 1} {}^qD = \exp\left(- \sum_{i=1}^{S} p_i \log p_i \right) \tag{2} $$

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:10 UTC from IEEE Xplore. Restrictions apply.

The order q determines the sensitivity of the metric to the relative species abundances, where typically measures for $q \in \{0, 1, 2\}$ are reported. For $q = 0$, the relative species abundances are not considered at all and $D(0)$ corresponds to the total number of species in the dataset (i.e., the richness). For $q = 1$, the species are weighted in proportion to their relative abundances, which is why $D(1)$ can be interpreted as the effective number of "typical" species. For $q = 2$, more weight is given to the most abundant species and the rarer species are discounted. As a result, $D(2)$ can be interpreted as the effective number of "common" or dominant species.

3 APPROACH

In this section, we first describe our approach to extend generator-based fuzzing with the ability to produce more behaviorally diverse inputs (Section 3.1–3.3). Then, we introduce a novel metric to quantify the behavioral diversity of a fuzzing campaign based on Hill numbers (Section 3.4).

3.1 Overview

The key idea of our approach can be summarized as follows: (i) We search for interesting *input structures* in the space of valid inputs through *structure-changing* mutations. (ii) We produce different variants of the same input structure by applying *structure-preserving* mutations with the goal of exploring diverse execution traces. To bias the input generation towards high behavioral diversity, we observe the coverage and validity feedback from the program after every execution and adapt the mutation strategy accordingly.

3.2 Structural Parameter Splitting

As described in Section 2.1, the non-determinism of the input generation process is entirely controlled by the sequence of random choices made. Our first insight is that the choices can be classified into two different types: *structural choices* and *value choices*, depending on their influence on the control-flow behavior of the generator. Based on the notion of a *choice point* by Reddy et al. [35], we define these choice types as follows:

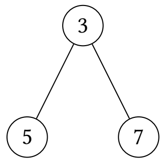
Definition 3.1 (Structural and Value choices). Let the choice point p be a tuple (ℓ, C), where $\ell \in \mathbb{L}$ is a program location in the generator G and $C \subseteq \mathbb{C}$ is a finite domain of choices. The choice point p is said to produce *structural choices* if the evaluation of a branch condition at some point during the execution of G depends on the choice $c \in C$. Otherwise, p is said to produce *value choices.*

For example, the choice point p_1 = (Line 2, `[0, 1, ..., 10]`) in Figure 2 produces value choices, since no branch condition in `generate_tree` depends on that choice. On the other hand, both choice points p_2 = (Line 6, `[True, False]`) and p_3 = (Line 8, `[True, False]`) produce structural choices, since the corresponding choices directly influence the branching behavior of the generator G (i.e., the decision on whether to generate child nodes or not).

Based on this insight, our first idea is to *split* the sequence of untyped parameters into two distinct parameter sequences based on the choice type they are used for:

(1) A *structural parameter sequence* to be consumed by structural choice points.
(2) A *value parameter sequence* to be consumed by value choice points.

That is, we can now represent each input as a tuple (σ_s, σ_v), where σ_s and σ_v represent the structural and value parameters consumed by the input generator, respectively. To further illustrate this idea, consider the following binary tree:



The corresponding parameter sequence σ (shown as typed choice values for the sake of simplicity) is given by: σ = 3, `True`, 5, `False`, `False`, `True`, 7, `False`, `False`

Here, the structural choices consists of all the boolean and the value choices of all the integer values. Therefore, we can represent this parameter sequence by the following two distinct sequences: σ_s = `True`, `False`, `False`, `True`, `False`, `False` and σ_v = 3, 5, 7.

This change in how the parameters for a generator are handled is a key idea of our approach, since it gives us access to the following operations and concepts.

First, it enables us to perform more controlled mutations on the input. In particular, by mutating the structural parameter sequence, we can directly mutate the structural choices made during the input generation process. By definition, the change of structural choices eventually results in a change of the control-flow behavior in the generator, which typically produces an input with a different structure. For example, if we consider the binary tree generator from Figure 2, mutating the structural parameters changes the boolean decisions on the generation of child nodes. On the other hand, by mutating the value parameters only, the control-flow behavior of the generator is preserved, yet different value choices are sampled. In the case of the binary tree generator, mutating value parameters results in mutated choices for the node values, while keeping the shape of the binary tree unmodified. Overall, the access to these structure-changing and structure-preserving mutations allows us to explore the input space in a more controlled manner.

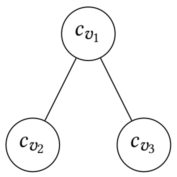
The second benefit of separating the structural and value parameters is that it allows us to synthesize an *abstract input* from a concrete input:

Definition 3.2 (Abstract Input, Structural Signature). Let i be an input generated by a generator G, $\sigma_s = c_{s_1}, c_{s_2}, \ldots, c_{s_n}$ be the corresponding sequence of structural parameters, and $\sigma_v = c_{v_1}, c_{v_2}, \ldots, c_{v_m}$ be the corresponding sequence of value parameters made by G during the generation of i. The *abstract input* $A(i)$ of the input i is the input that is obtained by setting all value parameters in σ_v as unspecified (or symbolic), while fixing the concrete structural parameters σ_s. The sequence of concrete structural parameters σ_s is called the *structural signature* of $A(i)$.

Intuitively, the abstract input represents the structural skeleton of the input, which may be concretized by specifying the missing value parameters. In our running example, the abstract input of a generated binary tree would correspond to a binary tree with the same shape but unspecified node values. As an example, for

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:10 UTC from IEEE Xplore. Restrictions apply.

the above described tree, the abstract input $A(t)$ fixes the structural parameters σ_s and leaves the value parameters σ_v unspecified ($c_{v_1}, c_{v_2}, c_{v_3}$), i.e.:



$$\sigma_s = \texttt{True}, \texttt{False}, \texttt{False}, \texttt{True}, \texttt{False}, \texttt{False}$$
$$\sigma_v = c_{v_1}, c_{v_2}, c_{v_3}$$

The structural signature (i.e., the sequence of structural parameters σ_s) of $A(t)$ thus describes the set of all binary trees consisting of three nodes: a root node that is connected to two child nodes where each node has a value between 0 and 10 (the domain of the choice point p_1 = (Line 2, `[0, 1, ..., 10]`)). Since we represent each input as a tuple (σ_s, σ_v), we can easily check whether two inputs share the same abstract input by comparing their structural signatures. This is another crucial aspect of our approach, as this property allows us to reason about the explored input space on a higher level. In particular, our approach uses the concept of abstract inputs in order to identify interesting input structures that exercise new behavior.

3.3 Feedback-Driven Search Strategy

The fuzzing algorithm of BeDivFuzz is presented in Algorithm 1. It is based on the coverage-guided Zest algorithm for generator-based fuzzing, and is extended by integrating the concepts of structural parameter splitting and abstract input structures. In particular, the two key components of the BeDivFuzz algorithm consist of:

(1) An adaptive mutation strategy that biases input generation towards high behavioral diversity, and
(2) A fuzzing heuristic that guides the search strategy based on the structural properties of the input.

We highlight these main innovations in Algorithm 1 in grey.

Algorithm 1: BeDivFuzz Algorithm

```
Input  : program $p$, generator $g$
Output : a set of test inputs, a set of failing inputs
1  $Q \leftarrow \{\text{RANDOM}\}$                    /* Initial seed input */
2  $F \leftarrow \emptyset$                              /* Failing parameters */
3  $C \leftarrow \emptyset$                                 /* Total coverage */
4  $T \leftarrow \emptyset$                                   /* Unique traces */
5  $S \leftarrow \emptyset$               /* Interesting valid input structures */
6  repeat
7      for $(\sigma_s, \sigma_v)$ in $Q$ do
8          for $1 \le i \le$ numChildren$(\sigma_s, \sigma_v)$ do
9              $\tilde{\sigma}_s, \tilde{\sigma}_v \leftarrow$ mutateAdaptive$(\sigma_s, \sigma_v)$
10             $input \leftarrow g(\sigma_s, \sigma_v)$
11             $coverage, result \leftarrow$ run$(p, input)$
12             if $result$ = Failure then                /* Save failures */
13                 $F \leftarrow F \cup \{(\sigma_s, \sigma_v)\}$
14             else /* Check if input has interesting structure */
15                 if $result$ = Valid and $coverage \not\subseteq C$ and $\sigma_s \notin S$ then
16                     $Q \leftarrow Q \cup \{(\sigma_s, \sigma_v)\}$        /* Add to queue */
17                     $S \leftarrow S \cup \{\sigma_s\}$       /* Add saved structure */
18                 updateCoverageStats$(coverage, result, C, T)$
19 until given time budget expires
20 return $g(Q), g(F)$
```

Similar to Zest (and other coverage-guided fuzzers), we maintain a queue Q of interesting inputs, which is initially seeded with a random value (Line 1). While Zest operates on single parameter sequences σ, BeDivFuzz operates on tuples (σ_s, σ_v) of split parameter sequences. For the sake of brevity, in the remainder of this paper we will use the term *parameters* when referring to tuples (σ_s, σ_v) of split parameter sequences. Throughout the fuzzing campaign, the algorithm maintains basic bookkeeping data, such as the set of failing parameters and the current branch coverage (Lines 2–3). Additionally, BeDivFuzz keeps track of all unique coverage traces (Line 4) and the set of interesting abstract input structures (Line 5). The set of unique coverage traces is used to adaptively bias the mutation strategy, described in more detail at the end of this section. The set of abstract input structures is utilized by BeDivFuzz to guide the search within the space of valid inputs.

The main fuzzing loop (Lines 6–19) has the same structure as other coverage-guided fuzzers, described as follows: First, a parameter sequence is selected from the queue (Line 7), after which the number of child parameters is determined (Line 8). We use the same heuristic as Zest, which computes the number of child parameters based on the coverage of the parent input. Each child is generated by performing one or more mutations on the parent parameter sequence (Line 9). The mutated parameters are then used to generate a concrete input using the provided generator (Line 10). Afterwards, the system under test is executed with the generated input, which yields the code coverage and the validity feedback (Line 11). The validity feedback is stored in the variable *result* and can be any of {Valid, Invalid, Failure}. In the case of Failure, the current parameters are saved to the set of failures (Line 13). Otherwise, the algorithm heuristically decides whether the current parameters should be saved to the queue (i.e., if the parameters are interesting enough to be further mutated) based on the observed execution results (Line 15). This is by default the case if the input is (1) valid, (2) exercises new coverage, and (3) represents an input structure that has not been previously added to the queue. Finally, the bookkeeping-data is updated (Line 18). This includes updating the branch coverage and the set of unique coverage traces.

As described in Section 3.2, BeDivFuzz is able to distinguish between *structure-changing* and *structure-preserving* mutations. This is done by performing mutations on either the structural parameters σ_s or on the value parameters σ_v. Performing structure-changing mutations allows to search for new program behavior that is only triggered if the input satisfies particular structural properties. In contrast, structure-preserving mutations allow to test a specific behavior of the code with different variants of the input (i.e., inputs with the same abstract structure but different values).

In order to decide which type of mutation to perform, BeDivFuzz keeps track of the types of mutations that have been performed on a parameter sequence and observes whether the resulting input has resulted in the execution of a *unique trace*. The following mutations are then biased towards the mutation operator that is more likely to produce unique execution traces, as illustrated in Algorithm 2. Specifically, with a probability of ϵ, a random type of mutation is performed. Otherwise, the mutation type is chosen based on heuristic scores R_s and R_v for the structural and value mutations, respectively (Line 6). For a given mutation type x, the score is calculated as the fraction of inputs that have exercised a unique

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:10 UTC from IEEE Xplore. Restrictions apply.

trace (U_x) out of all the inputs that were generated by the respective mutation type (N_x), i.e.:

$$R_x = \frac{U_x}{N_x} \quad x \in \{s, v\} \tag{3}$$

Thus, we bias the selection of the mutation operator towards producing inputs that diversely exercise particular behaviors, while relying on the structural search heuristic in Algorithm 1 (Line 15) to discover new behaviors.

Algorithm 2: Adaptive Mutation Strategy

```
Input  : structural parameters $\sigma_s$, value parameters $\sigma_v$, exploration factor $\epsilon$
Output : mutated parameters $\tilde{\sigma}_s$, $\tilde{\sigma}_v$
1  if uniformRandom() < $\epsilon$ then
2      if randomBoolean() then         /* Mutate either of the params */
3          return (mutate($\sigma_s$), $\sigma_v$)
4      else
5          return ($\sigma_s$, mutate($\sigma_v$))
6  $R_s, R_v \leftarrow$ calculateScores() /* Score by mutation type (Eq. 4) */
7  if $R_s \neq R_v$ then              /* Perform most promising mutation */
8      if $R_s > R_v$ then
9          return (mutate($\sigma_s$), $\sigma_v$)   /* Mutate structural params */
10     else
11         return ($\sigma_s$, mutate($\sigma_v$))        /* Mutate value params */
12 else
13     goto 2                         /* Resort to random mutation */
```

3.4 A Metric for Behavioral Diversity

As part of this work, we are interested in establishing a notion of *behavioral diversity* in the context of fuzz testing. In particular, we propose to use *Hill numbers* (Section 2.2), or *effective number of species* [17]. The inspiration for this metric stems from the STADS (*Software Testing and Analysis as Discovery of Species*) framework introduced by Böhme [6]. In this framework, Böhme links the sampling process in an assemblage for species discovery to the process of sampling from the program's input space to discover particular features of the input's execution (e.g., covered branches, reached program states). However, rather than using the framework to estimate species discovery probabilities [6, 7], we build upon the same connection between ecology and software testing by applying an established biodiversity index — known as Hill numbers — to quantify the behavioral diversity of a fuzzing campaign.

We apply this diversity measure to the context of fuzzing as follows. Consider a fuzzing campaign, where a fuzzer F samples inputs I to fuzz a program P. Executing P with an input $i \in I$ results in a trace $t(i)$ that consists of the sequence of branches visited during the execution. Each branch b that has been covered by at least one input i can now be seen as a species. The abundance of a species (i.e., of a covered branch b) can then be computed as the number $c(b)$ of unique traces that have executed the branch b (the branch execution count). Given these measures, we can now quantify the *behavioral diversity* of a fuzzing campaign as the Hill numbers computed over the distribution of branch execution counts.

Definition 3.3 (Behavioral Diversity). Let I be the set of inputs generated by a fuzzer during a fuzzing campaign, and let $C : \mathcal{B} \mapsto \mathbb{N}_{>0}$ be a function that maps a covered branch $b \in \mathcal{B}$ to its relative execution count over all unique traces. Then, the *behavioral diversity* of order q is defined as:

$$B(q) = \left(\sum_{b \in \mathcal{B}} C(b)^q \right)^{1/(1-q)} \tag{4}$$

Intuitively, this measure quantifies the *effective number* of diversely covered branches, that is, the number of branches that were equally often executed by diverse inputs (i.e., inputs with unique traces). While $B(0)$ is equal to the number of covered branches (since it equally weights rare and common branches), $B(1)$ weights branches by their relative execution counts and can thus be interpreted as the "effective number of typically covered branches". $B(2)$ gives more weight to common branches and can be seen as the "effective number of common branches".

In the context of fuzzing, behavioral diversity considers the distribution of diverse branch execution counts in order to measure the degree of exploration bias towards specific program behavior. Higher numbers indicate that a fuzzer is diversely exploring more branches, whereas low numbers mean that the fuzzer is spending most of the resources exploring the same few behaviors. Low behavioral diversity is most evident for random blackbox-techniques, which may be able to spuriously cover many branches, but ultimately exercise the same likely code regions over and over again. Especially for "unsuccessful" fuzz campaigns where no bugs are found, behavioral diversity can provide valuable insights into the progress of a fuzzer as a complementary metric to branch coverage. For instance, practitioners may consider more diversely executed branches as more reliable than less diversely executed ones, since they have been tested more "thoroughly". On the other hand, researchers may tune their fuzzers to repeatedly target less diversely executed branches in order to increase the overall confidence in the reliability of the respective behavior.

4 EVALUATION

In this section, we evaluate the effectiveness of BeDivFuzz in generating inputs that exhibit diverse behavior. We compare our approach against three techniques: RLCheck [35], Zest [32], and an implementation of Quickcheck [12] from the JQF [31] fuzzing framework. In particular, we seek to answer the following research questions:

- RQ1 Is BeDivFuzz able to effectively produce diverse valid inputs for real-world benchmarks? (Section 4.2)
- RQ2 Do the inputs generated by BeDivFuzz have a higher behavioral diversity compared to state-of-the-art techniques? (Section 4.3)
- RQ3 How does BeDivFuzz perform in terms of fault finding capabilities? (Section 4.4)

4.1 Study Design

Baseline Techniques. We compare against two state-of-the-art techniques in generator-based testing, namely RLCheck [35] and Zest [32]. RLCheck uses reinforcement learning in order to automatically learn a *guide* that leads the generator towards producing many diverse valid inputs. Zest is a generator-based fuzzer that integrates validity feedback into a coverage-guided search algorithm to effectively cover the semantic analysis code. We also compare against a Java version of Quickcheck [12] by running Zest without code coverage and validity feedback, as done in the original evaluation

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:10 UTC from IEEE Xplore. Restrictions apply.

of RLCheck. That is, Quickcheck performs random sampling of inputs using the generator.

Experimental Subjects. Our evaluation is conducted on six real-world benchmarks, namely Apache Ant, Apache Maven, Mozilla Rhino, Google Closure Compiler, Oracle Nashorn, and Apache Tomcat. The first four subjects have been used in the original evaluations of Zest [32] and RLCheck [35]; we add two additional subjects for a broader benchmark. In addition, we have updated the subjects to the latest versions available at the time when we conducted the experiments. Inputs for Ant, Maven, and Tomcat are generated by an XML generator, whereas Rhino, Closure, and Nashorn use a JavaScript code generator.

Configuration. We run all baseline techniques with their default configurations. In order to run Nashorn and Tomcat with RLCheck, we use configuration files based on those provided for Rhino and Maven by the original authors, respectively. For BeDivFuzz, we use $\epsilon = 0.2$ as the probability to perform a random mutation. Further, we provide results for two configurations of BeDivFuzz, which we refer to as BeDivFuzz-structure and BeDivFuzz-simple. BeDivFuzz-structure utilizes both structural mutations and analysis of input structure novelty, whereas BeDivFuzz-simple only employs structural mutations. In order to make the input generators used by Zest compatible with our technique, we manually extended them to handle split parameter sequences and annotated the choice points with the type of choice they produce. However, we note that the latter could also be done automatically through data-flow analysis.

Implementation. We have implemented BeDivFuzz [4] as an extension of Zest in JQF [31]. JQF is a framework for generator-based testing in Java and implements different fuzzing algorithms, including the Zest algorithm and AFL-like fuzzing for Java programs.

Experimental Parameters. For the evaluation of RQ1 (input diversity) and RQ2 (behavioral diversity), we run each experiment with a timeout of 1 hour. The original evaluation of RLCheck [35] was conducted with a timeout of 5 minutes, as the authors assumed a property-based testing context where the generator is typically executed only for a short time. While a 5 minute timeout may seem short compared to common timeouts in fuzzing evaluations, it should also be noted that collecting coverage metrics for blackbox methods (e.g., Quickcheck or RLCheck) is significantly more expensive than for greybox methods (e.g., Zest or BeDivFuzz). In particular, the inputs for blackbox techniques first need to be generated on the uninstrumented SUT until timeout, and subsequently be replayed on the instrumented version of the SUT. This is necessary to allow for a fair comparison against greybox methods, since instrumentation adds additional overhead. However, this methodology results in significantly longer experimental runtimes[1]. On the other hand, coverage data may not be meaningful if timeouts are too short, since coverage-guided algorithms typically need some time to become effective. Thus, for the evaluation of RQ1 (input diversity) and RQ2 (behavioral diversity), we extend the timeout for each experiment to 1 hour. We further justify this decision by our observation that coverage has plateaued in almost all experiments after this timeout. However, for RQ3 (fault finding capabilities), we extend the timeout to 24 hours, since evaluations that focus on bug metrics should be performed with longer timeouts, as suggested by Klees et al. [23]. To account for the variability of the results due to randomness, we perform 30 repetitions. Statistical significance is assessed using the Mann-Whitney U test (also known as the Wilcoxon rank-sum test) with $\alpha = 0.01$, as suggested by Arcuri and Briand [1] for randomized algorithms.

As for hardware, we conducted all experiments on a server with an Intel(R) Xeon(R) E7-4880 2.5GHz CPU and 1TB of RAM running openSUSE Leap 15.

[1]In our experiments, we have observed up to 50× longer runtimes when computing coverage for Quickcheck and RLCheck.

4.2 RQ1: Generating Diverse Valid Inputs

We answer RQ1 by assessing the number of diverse valid inputs generated by BeDivFuzz compared to the baseline techniques. Similar to Reddy et al. [35], we consider *diverse valid inputs* as inputs that exercise different traces, rather than inputs that are only unique on the byte or string level. The results are visualized in Figure 3. The left column shows for each benchmark subject the *percentage* of all generated inputs that were diverse valid inputs, whereas the right column depicts the *total* number of diverse valid inputs.

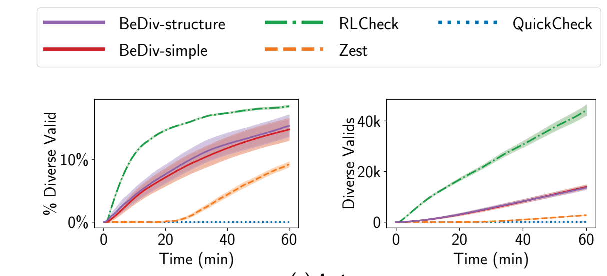
If we consider the percentages of generated diverse valid inputs, the results indicate that BeDivFuzz is competitive with the current state-of-the-art. For all benchmark subjects, both configurations of BeDivFuzz outperform Zest and Quickcheck as they are able to maintain a high percentage of diverse valid inputs and even increase it over time for several subjects. However, in two out of six subjects, RLCheck significantly outperforms BeDivFuzz, which is most notable in the Maven benchmark where the mean percentage of diverse valid inputs is about 6% and 10% higher compared to BeDivFuzz-structure and BeDivFuzz-simple, respectively.

The plots for Ant show a high variability in the performance of BeDivFuzz, which is due to the difficulty of finding the first valid input for this subject. This indicates that BeDivFuzz can potentially benefit from an initial valid seed input. The same applies to Zest, which also further mutates invalid inputs by default, thus potentially wasting resources by initially exploring the error handling code of the SUT. On the other hand, the efficiency of RLCheck allows it to quickly find a valid input and exploit the obtained information to generate further diverse valid inputs.

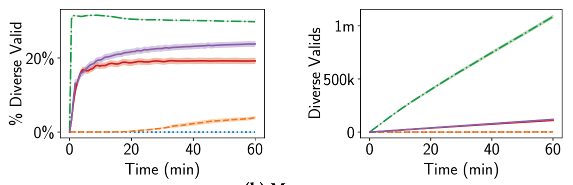
On our newly added subjects Nashorn and Tomcat, RLCheck does not perform as well as on the other benchmarks. This can potentially be attributed to the fact that the new subjects have very simple validity functions, while different behavior may only be triggered by specific input structures. We assume that since RLCheck only relies on the validity feedback, it is unable to learn any meaningful policy as almost any generated input is valid, but not necessarily diverse valid. As a result, RLCheck might be prone to overfit to a valid, yet uninteresting space of inputs (with regard to the triggered program behavior). In contrast to that, BeDivFuzz also utilizes code coverage to identify whether an input with a unique structure exercises interesting behavior or not.

While the results indicate that BeDivFuzz is highly effective in generating diverse valid inputs, the total numbers (Figure 3, right column) show that our approach is mostly limited by its efficiency. Overall, BeDivFuzz only performs better than Zest in

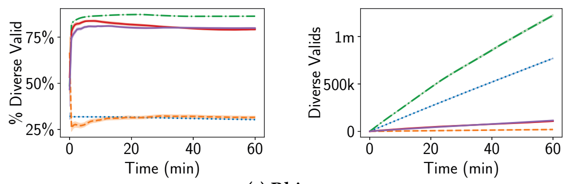
Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:10 UTC from IEEE Xplore. Restrictions apply.



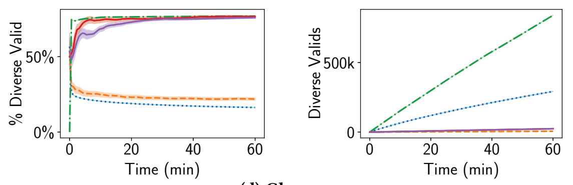
(a) Ant



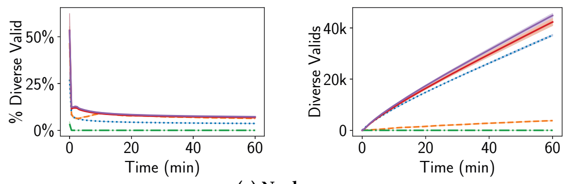
(b) Maven



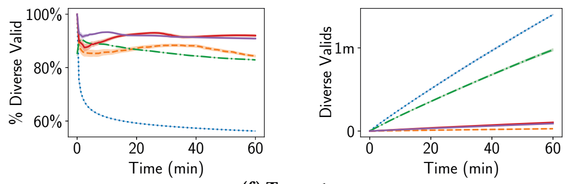
(c) Rhino



(d) Closure



(e) Nashorn



(f) Tomcat

Figure 3: Percent (left) and absolute number (right) of diverse valid inputs (i.e., valid inputs with different traces).

terms of generating many diverse valid inputs. When comparing both configurations of BeDivFuzz, BeDivFuzz-structure tends to have a slight edge due to the added analysis of input structures, but the differences are generally not significant. However BeDivFuzz, is outperformed by both RLCheck and Quickcheck for most of the benchmark subjects. This can be explained as follows. Like Zest, BeDivFuzz requires the SUT to be instrumented in order to collect code coverage. This results in a significant slowdown of execution time (around 10–100× slower), effectively reducing the total number of diverse valid inputs that can potentially be generated. On the other hand, while RLCheck has a comparable effectiveness to BeDivFuzz, the blackbox approach makes it much more efficient. Nevertheless, this is an acceptable trade-off, since our main focus is not on producing a large number of diverse valid inputs. Instead, the overall goal of BeDivFuzz is to exercise *diverse behavior*, which we evaluate in the following section.

4.3 RQ2: Diverse Execution of Program Behavior

In this section, we seek to answer RQ2, that is, whether BeDivFuzz is able to generate test inputs that have a higher behavioral diversity compared to the baseline techniques. We measure the behavioral diversity by the *behavioral diversity index* $B(q)$ of order q, defined in Equation 4. In particular, we compare the behavioral diversity indices for $q \in \{0, 1, 2\}$, since in the field of ecology, Hill numbers are usually reported for these orders as well. Recall that the behavioral diversity $B(q)$ can be interpreted as follows:

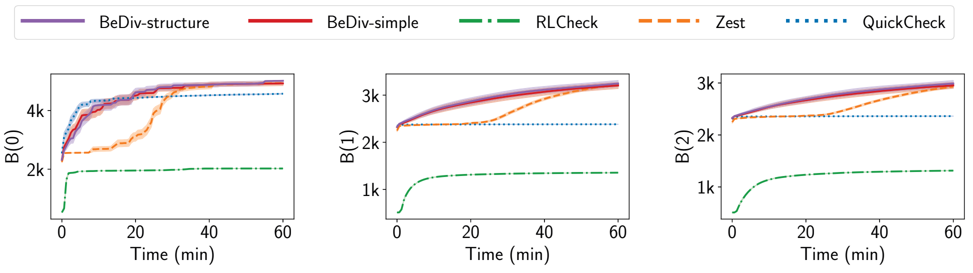
$q = 0$: The total number of covered branches (i.e., branch coverage)
$q = 1$: The effective number of typically executed branches
$q = 2$: The effective number of commonly executed branches

The *effective number* of a set of covered branches can be seen as the number of branches executed by the proportionally same number of diverse inputs. We are mostly interested in the results for $B(1)$ and $B(2)$, since they emphasize the relative execution counts of the typical and more common branches, respectively.

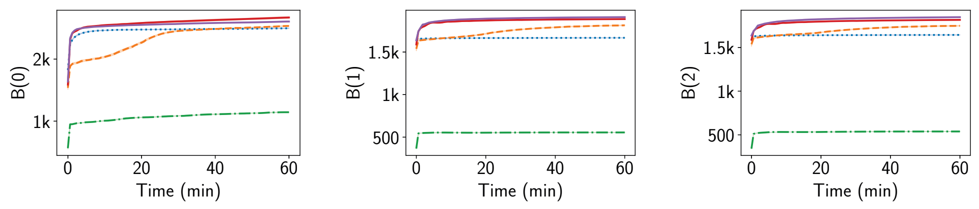
The results are shown in Figure 4. From left to right, the columns depict the behavioral diversity of increasing order q (from 0 to 2). The first column shows for each subject and technique the behavioral diversity of order 0, i.e., the total number of covered branches. While RLCheck is able to generate the highest number of diverse valid inputs (Section 4.3), it performs the worst in terms of branch coverage. When comparing the other approaches, both configurations of BeDivFuzz have covered the highest number of branches in three out of six benchmarks (Ant, Maven, Nashorn), with BeDivFuzz-simple additionally outperforming all other techniques in Tomcat. However, Quickcheck outperforms BeDivFuzz in the two remaining benchmarks (Rhino and Closure). Comparing against Zest, BeDivFuzz is only significantly outperformed in Closure, and also in Tomcat for BeDivFuzz-structure.

The second column of Figure 4 compares the different techniques w.r.t. the $B(1)$ metric, i.e., the effective number of "typically" executed branches. Here, a similar trend in the results can be observed. While the relative performance of RLCheck remains the same, both configurations of BeDivFuzz perform significantly better than all baseline approaches in four out of six benchmarks (Maven, Rhino, Closure, Nashorn). Most interestingly, when comparing against the

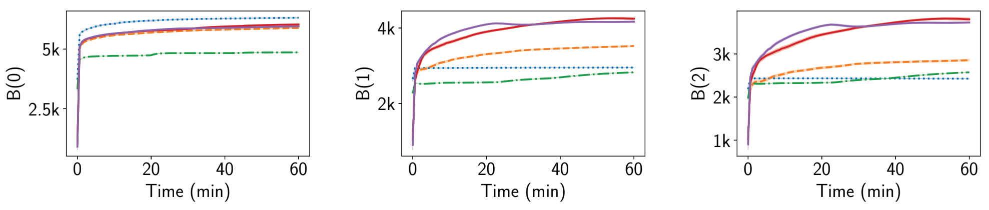
Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:10 UTC from IEEE Xplore. Restrictions apply.



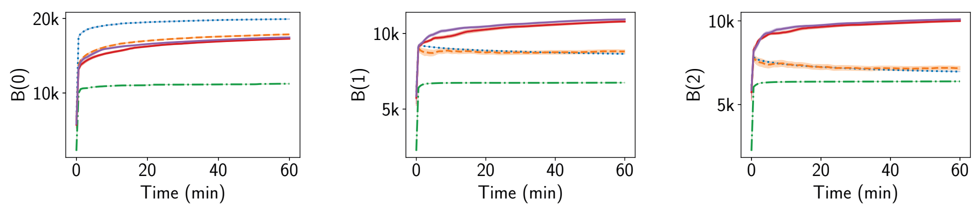
(a) Ant



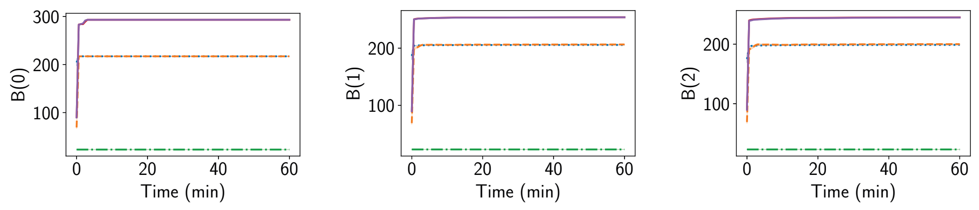
(b) Maven



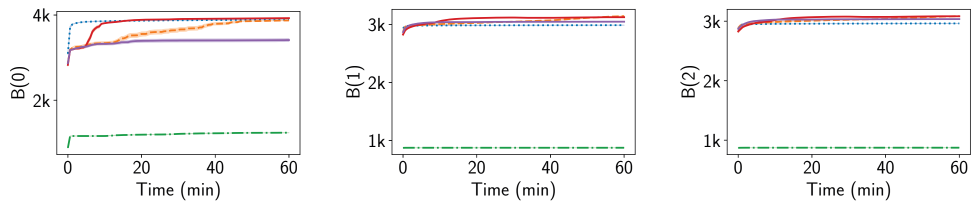
(c) Rhino



(d) Closure



(e) Nashorn



(f) Tomcat

Figure 4: From left to right: Behavioral diversity of increasing order 0 to 2.

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:10 UTC from IEEE Xplore. Restrictions apply.

plots of the first column ($B(0)$, i.e., number of covered branches), BeDivFuzz now outperforms both Quickcheck and Zest in Rhino and Closure. In the remaining benchmarks (Ant and Tomcat), the performance of Zest is comparable to BeDivFuzz. Another observation is that the $B(1)$ value of BeDivFuzz tends to increase over time, most noticeable on Ant, Rhino, and Closure. This indicates that the inputs generated by BeDivFuzz not only diversely execute a fixed set of behaviors. Instead, BeDivFuzz is able to continuously increase the coverage of new behaviors while maintaining the overall diversity of the triggered behavior. In contrast, the $B(1)$ metric even slightly decreases over time for Quickcheck in the Closure benchmark. A possible explanation for this observation is that Quickcheck is prone to disproportionately often trigger the most likely behaviors in the SUT. On the other hand, most of the other covered branches may have been executed only a few times by chance. This increasing "unevenness" in executed behavior could consequently result in a decline of the $B(1)$ value.

The third column shows the results for the $B(2)$ metric, i.e., the effective number of commonly executed branches. Again, in four out of six benchmarks (Maven, Rhino, Closure, Nashorn), BeDivFuzz performs significantly better than all baseline approaches. For the Rhino and Closure benchmark, the difference between BeDivFuzz and the baselines even increased compared to the results of the $B(1)$ metric (center column). That is, both configurations of BeDivFuzz generate inputs that trigger "common" behavior in a more diverse manner compared to the baselines.

Throughout the experiments, when comparing the two configurations of BeDivFuzz, we have observed that BeDivFuzz-structure generally performs better for shorter timeouts (≤ 5 minutes). Since BeDivFuzz-structure only saves inputs to the queue if they exhibit a new input structure, the queue of saved inputs contains less, but more (structurally) diverse inputs. As a result, BeDivFuzz-structure initially explores more diverse behavior, but the plain adaptive mutation strategy of BeDivFuzz-simple seems to be sufficient to eventually discover these behaviors as well. Thus, the more explorative nature of BeDivFuzz-structure may be more suitable for contexts where runtimes are required to be short (e.g., property-based testing).

4.4 RQ3: Finding Faults

Table 1 shows the list of crashes that were discovered after a timeout of 24 hours, deduplicated by benchmark and exception type (as done in the evaluations of Zest and RLCheck). To answer RQ3, we compare the different approaches concerning the discovery times and the reliability of triggering a particular crash. The results indicate that RLCheck performs the worst in terms of fault finding capabilities, as this approach has triggered the least number of crashes. In contrast, Zest performs the best as it has found one additional crash compared to BeDivFuzz and Quickcheck, though this crash was only found in 16% of the trials. This is is most likely due to the coverage-guided algorithm of Zest that allows to effectively explore deeper paths in the semantic analysis stage compared to Quickcheck. On the other hand, while BeDivFuzz also leverages coverage guidance, the approach focuses more on diversely executing many different behaviors through controlled mutations. Thus, BeDivFuzz may miss interesting edge cases that are more likely to be produced by completely random mutations.

Comparing the metrics for the bugs found by BeDivFuzz, Zest, and Quickcheck, it can be noted that Quickcheck finds the crashes faster and more reliably. The crashes in Closure and Rhino are typically found within minutes by Quickcheck, while BeDivFuzz and Zest require hours to discover most of the crashes. This result can potentially be explained by results in Section 4.3, which show that Quickcheck already achieved high coverage after a few minutes in these subjects. In contrast, the search algorithms of BeDivFuzz and Zest might have prioritized exploring other parts of the program, before eventually discovering the crash-inducing behaviors.

Interestingly, BeDivFuzz-structure tends to find crashes faster than BeDivFuzz-simple, which suggests the importance of the structural novelty heuristic for fault finding.

4.5 Threats to Validity

Internal Validity. To avoid potential systematic errors that could pose threats to internal validity, we have designed our experiments (replication count, timeout, etc.) based on the guidelines provided by Klees et al. [23]. Additionally, we have reused existing and available implementations of the baseline fuzzers with conforming parameter settings from the evaluation of RLCheck [35]. The parameters for BeDivFuzz are not tuned; thus we can provide a fair and realistic comparative evaluation of the different approaches.

External Validity. Our evaluation focuses only on six programs (Ant, Maven, Rhino, Closure, Nashorn, and Tomcat) with two different input formats, namely XML and JavaScript. Whether our results can be generalized to other programs and other input formats is a threat to external validity. However, the programs under test represent complex, long-living, and mature programs with a widespread adoption. Thus, we argue that an application of BeDivFuzz to similar programs would produce similar results.

Construct Validity. The main question towards construct validity is whether the Hill numbers [17] for species diversity are actually a good metric to evaluate the behavioral diversity of the covered branches in a fuzzing campaign. We argue that behavioral diversity is given not only if many different behaviors are covered (i.e., a high branch coverage), but each of the different behaviors is equally covered by many diverse inputs. Utilizing Hill numbers as a metric specifically accounts for the possible variations in the diverse execution of different behaviors, which may differ greatly depending on the chosen fuzzing technique as shown in our evaluation.

5 RELATED WORK

Diversity in Fuzzing and Search-based Test Case Generation There have been several fuzzing and search-based test case generation approaches that target diversity as one of their objectives. These approaches can be classified into approaches that a) aim to achieve diversity in the input space [5, 11, 28, 39, 46] or b) in the covered behavior of the program under test when executing the test cases. One of the earlier approaches in the first category is Adaptive Random Testing (ART) [11], a black box testing approach that aims to distribute the test cases over the entire input space. Technically, ART selects the next test case that maximizes the minimal distance to all already executed test cases. A similar approach is applied in

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:10 UTC from IEEE Xplore. Restrictions apply.

Crash-ID	BeDiv-simple	BeDiv-structure	Zest	Quickcheck	RLCheck
closure.StringIndexOutOfBoundsException	645 (76%)	473 (73%)	82 (100%)	3 (100%)	-
closure.NullPointerException	425 (100%)	347 (100%)	149 (100%)	4 (100%)	7 (100%)
closure.RuntimeException	121 (100%)	88 (100%)	5 (100%)	<1 (100%)	6 (100%)
rhino.ArrayIndexOutOfBoundsException	-	-	960 (16%)	-	-
rhino.ClassCastException	650 (40%)	371 (33%)	606 (56%)	192 (100%)	-
rhino.IllegalStateException	1 (100%)	2 (100%)	1 (100%)	<1 (100%)	<1 (100%)
rhino.NullPointerException	11 (100%)	7 (100%)	5 (100%)	<1 (100%)	-
rhino.VerifyError	543 (80%)	452 (83%)	212 (100%)	6 (100%)	9 (100%)
nashorn.AssertionError	44 (100%)	48 (100%)	487 (66%)	125 (100%)	-

Table 1: Average time (in minutes) and reliability of triggering a particular crash.

the tool DIG [5] for testing web-based applications. The distance of test cases is computed based on the sequences of actions that traverse the navigational model of the web application. In search-based testing, Sapienz^{div} [46] aims to maintain and improve a set of test cases for mobile applications that trigger diverse inputs, measured by the distance of test input sequences. The "Uncommon inputs" strategy from Soremekun et al. [39] creates inputs based on an inverted probabilistic grammar. If the original probabilistic grammar is learned from a set of common samples, the hypothesis is that inverting the probabilities would lead to uncommon and more diverse inputs.

BeDivFuzz belongs to the second category of approaches since we focus on behavioral diversity and achieving input diversity as a byproduct. A first step towards behavioral diversity is taken in any greybox fuzzing approach (e.g., [2, 8, 9, 15, 24–26, 43, 53]) that aim to maximize some coverage metric in the SUT, such as branch or statement coverage. The feedback loop in these greybox fuzzing approaches implements a novelty search that values test inputs that provide additional coverage. This feedback loop is also present in hybrid fuzzing approaches (e.g., [34, 42, 52]) that combine greybox fuzzing with symbolic execution [10, 13, 22, 50] or concolic testing [36]. Furthermore, the general idea of greybox fuzzing is taken a step further in FairFuzz [24], VUzzer [34], and TortoiseFuzz [49]. FairFuzz is an AFL [53] extension with the goal of triggering rare branches. The main idea of FairFuzz is to learn mutation masks and areas that have a higher chance of hitting these rare branches. TortoiseFuzz and VUzzer instead prioritize exploration of code regions with a high chance of containing a vulnerability. In the case of VUzzer, the goal is to cover error-handling code, and TortoiseFuzz aims to cover memory access operations.

In contrast to all these approaches, we argue that just hitting a branch once does not provide behavioral diversity. The existing approaches basically aim to optimize the $B(0)$ metric (i.e., branch coverage) in Figure 4, whereas BeDivFuzz also provides behavioral diversity and scores well on the $B(1)$ and $B(2)$ metrics.

Fuzzing and Generation of Valid Inputs The generation of syntactically and semantically valid inputs has always been the target of modern fuzzing approaches. Concerning syntactically correct inputs, the common approach is to use models to describe the input structure. Examples are input specifications [20, 21, 44] or grammars [2, 14, 16, 18, 30, 39, 48]. However, having just syntactically correct inputs is often not enough, and to explore deeper regions of the SUT, semantic validity of the inputs is required. Zest [32] utilizes validity and code coverage feedback to produce inputs with high semantic coverage. BeDivFuzz is based on the same feedback mechanism as Zest, but extends it with novel structural mutation operators and a structure-aware fuzzing heuristic. RLCheck [35] uses reinforcement-learning to learn a policy to guide the generator towards high input diversity. Similarly, BeDivFuzz automatically adapts its mutation strategy based on the received feedback. While our approach is built on top of these approaches, the main distinguishing factor is that we also aim to produce diverse behavioral inputs to have a more systematic exploration of the SUT's behavior.

6 CONCLUSION AND FUTURE WORK

In this paper, we have described an approach to generate test inputs with high behavioral diversity. That is, our approach does not only aim to *cover* as many behaviors as possible, but also to *diversely execute* the different behaviors. The key to our approach is to distinguish between *structure-changing mutations* that allow to search for new behaviors triggered by specific input structures, and *structure-preserving* mutations to diversely execute a particular behavior. This method is complemented by an adaptive mutation strategy and a new fuzzing heuristic that is based on the structural novelty of an input. We implemented this approach in BeDivFuzz, and show that it outperforms the current state-of-the-art w.r.t. to a novel measure of behavioral diversity that is inspired by a popular biodiversity metrics in ecology — *Hill-numbers*. In future work, we would like to provide guarantees for our approach by evaluating measures such as the residual risk [7] and reliability of a SUT after we terminate a fuzzing campaign.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments and suggestions, which helped to improve this paper. This research was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – GR 3634/7-1 ProCI (421921612); GR 3634/6-1 FLASH (261444241); GR 3634/4-2 Emperor (392561203).

REFERENCES

[1] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250. https://doi.org/10.1002/stvr.1486
[2] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security*

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:10 UTC from IEEE Xplore. Restrictions apply.

Symposium, NDSS 2019. The Internet Society. https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/

[3] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. Ijon: Exploring Deep State Spaces via Fuzzing. In *2020 IEEE Symposium on Security and Privacy, SP 2020*. IEEE, 1597–1612. https://doi.org/10.1109/SP40000.2020.00117

[4] BeDivFuzz. 2022. Source code and replication package. https://github.com/hub-se/BeDivFuzz.

[5] Matteo Biagiola, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2019. Diversity-based web test generation. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE*. ACM, 142–153.

[6] Marcel Böhme. 2018. STADS: Software Testing as Species Discovery. *ACM Trans. Softw. Eng. Methodol.* 27, 2, Article 7 (June 2018), 52 pages. https://doi.org/10.1145/3210309

[7] Marcel Böhme, Danushka Liyanage, and Valentin Wüstholz. 2021. Estimating residual risk in greybox fuzzing. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 230–241. https://doi.org/10.1145/3468264.3468570

[8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. ACM, New York, NY, USA, 2329–2344. https://doi.org/10.1145/3133956.3134020

[9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing As Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. ACM, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[10] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[11] Tsong Yueh Chen, Hing Leung, and I. K. Mak. 2004. Adaptive Random Testing. In *Advances in Computer Science - ASIAN 2004, Higher-Level Decision Making, 9th Asian Computing Science Conference (Lecture Notes in Computer Science, Vol. 3321)*, Michael J. Maher (Ed.). Springer, 320–329. https://doi.org/10.1007/978-3-540-30502-6_23

[12] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, 268–279. https://doi.org/10.1145/351240.351266

[13] Lori A. Clarke. 1976. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering* SE-2, 3 (Sept 1976), 215–222. https://doi.org/10.1109/TSE.1976.233817

[14] Martin Eberlein, Yannic Noller, Thomas Vogel, and Lars Grunske. 2020. Evolutionary Grammar-Based Fuzzing. In *Proceedings of the 12th Symposium on Search-Based Software Engineering (SSBSE 2020)*.

[15] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018*. IEEE Computer Society, 679–696. https://doi.org/10.1109/SP.2018.00040

[16] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-Based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 206–215. https://doi.org/10.1145/1375581.1375607

[17] Mark Hill. 1973. Diversity and Evenness: A Unifying Notation and Its Consequences. *Ecology* 54 (03 1973), 427–432. https://doi.org/10.2307/1934352

[18] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. USENIX, Bellevue, WA, 445–458.

[19] Paul Holser. 2014. junit-quickcheck: Property-based testing, JUnit-style. https://pholser.github.io/junit-quickcheck. Accessed: August 26, 2021.

[20] William Johansson, Martin Svensson, Ulf E. Larson, Magnus Almgren, and Vincenzo Gulisano. 2014. T-Fuzz: Model-Based Fuzzing for Robustness Testing of Telecommunication Protocols. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014*. IEEE Computer Society, 323–332. https://doi.org/10.1109/ICST.2014.45

[21] Rauli Kaksonen, M. Laakso, and A. Takanen. 2001. System Security Assessment through Specification Mutations and Fault Injection. In *Communications and Multimedia Security Issues of the New Century, Proceedings of the IFIP TC6/TC11 International Conference on Communications and Multimedia Security Issues (IFIP Conference Proceedings, Vol. 192)*, Ralf Steinmetz, Jana Dittmann, and Martin Steinebach (Eds.). Kluwer.

[22] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

[23] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conf. on Comp. and Comm. Security* (Toronto, Canada) *(CCS '18)*. ACM, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[24] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the 33rd ACM/IEEE Int. Conf. on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 475–485. https://doi.org/10.1145/3238147.3238176

[25] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-state Based Binary Fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 627–637. https://doi.org/10.1145/3106237.3106295

[26] libFuzzer. 2020. A library for coverage-guided fuzz testing - LLVM 3.9 documentation. http://llvm.org/docs/LibFuzzer.html. Accessed: August 26, 2020.

[27] B. Liskov and J. Guttag. 1986. *Abstraction and specification in program development.* MIT Press Cambridge, MA, USA.

[28] Hector D. Menendez and David Clark. 2021. Hashing Fuzzing: Introducing Input Diversity to Improve Crash Detection. *IEEE Transactions on Software Engineering* (2021).

[29] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for c. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 245–258. https://doi.org/10.1145/1542476.1542504

[30] Hoang Lam Nguyen, Nebras Nassar, Timo Kehrer, and Lars Grunske. 2020. MoFuzz: A Fuzzer Suite for Testing Model-Driven Software Engineering Tools. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 1103–1115. https://doi.org/10.1145/3324884.3416668

[31] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: coverage-guided property-based testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 398–401. https://doi.org/10.1145/3293882.3339002

[32] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. ACM, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576

[33] Peach Tech. 2020. Peach Fuzzer Platform. https://www.peach.tech/products/peach-fuzzer/peach-platform/. Accessed: August 26, 2020.

[34] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/

[35] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. 2020. Quickly generating diverse valid test inputs with reinforcement learning. In *ICSE '20: 42nd International Conference on Software Engineering, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1410–1421. https://doi.org/10.1145/3377811.3380399

[36] Koushik Sen. 2007. Concolic Testing. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering* (Atlanta, Georgia, USA) *(ASE '07)*. ACM, New York, NY, USA, 571–572. https://doi.org/10.1145/1321631.1321746

[37] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[38] Claude E. Shannon. 1948. A mathematical theory of communication. *Bell Syst. Tech. J.* 27, 3 (1948), 379–423. https://doi.org/10.1002/j.1538-7305.1948.tb01338.x

[39] Ezekiel Soremekun, Esteban Pavese, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. 2020. Inputs from Hell: Learning Input Distributions for Grammar-Based Test Generation. *IEEE Transactions on Software Engineering* (2020).

[40] Ian F. Spellerberg and Peter J. Fedor. 2003. A tribute to Claude Shannon (1916–2001) and a plea for more rigorous use of species richness, species diversity and the 'Shannon–Wiener' Index. *Global Ecology and Biogeography* 12, 3 (2003), 177–179. https://doi.org/10.1046/j.1466-822X.2003.00015.x

[41] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015*, Kunle Olukotun, Aaron Smith, Robert Hundt, and Jason Mars (Eds.). IEEE

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:10 UTC from IEEE Xplore. Restrictions apply.

Computer Society, 46–55. https://doi.org/10.1109/CGO.2015.7054186

[42] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

[43] Robert Swiecki and Felix Gröbert. 2021. honggfuzz. https://github.com/google/honggfuzz. Accessed: August 26, 2021.

[44] Peach Tech. 2018. Peach Fuzzer. https://www.peach.tech/products/peach-fuzzer/. Accessed: 2018-01-28.

[45] Luca Della Toffola, Cristian-Alexandru Staicu, and Michael Pradel. 2017. Saying 'hi!' is not enough: mining inputs for effective test generation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 44–49. https://doi.org/10.1109/ASE.2017.8115617

[46] Thomas Vogel, Chinh Tran, and Lars Grunske. 2021. A comprehensive empirical evaluation of generating test suites for mobile applications with diversity. *Inf. Softw. Technol.* 130 (2021), 106436. https://doi.org/10.1016/j.infsof.2020.106436

[47] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 579–594. https://doi.org/10.1109/SP.2017.23

[48] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*. IEEE / ACM, 724–735. https://doi.org/10.1109/ICSE.2019.00081

[49] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020*. The Internet Society.

[50] Guowei Yang, Suzette Person, Neha Rungta, and Sarfraz Khurshid. 2014. Directed Incremental Symbolic Execution. *ACM Trans. Softw. Eng. Methodol.* 24, 1 (2014), 3:1–3:42. https://doi.org/10.1145/2629536

[51] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. 2021. Automated conformance testing for JavaScript engines via deep compiler fuzzing. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 435–450. https://doi.org/10.1145/3453483.3454054

[52] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium, USENIX Security 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[53] Michal Zalewski. 2021. American Fuzzy Lop (AFL) - a security-oriented fuzzer. http://lcamtuf.coredump.cx/afl/. Accessed: August 26, 2021.

[54] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning. In *29th USENIX Security Symposium, USENIX Security 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2255–2269. https://www.usenix.org/conference/usenixsecurity20/presentation/zong

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:10 UTC from IEEE Xplore. Restrictions apply.