The Havoc Paradox in Generator-Based Fuzzing (Registered Report)

Ao Li
Carnegie Mellon University
Pittsburgh, USA
aoli@cmu.edu

Madonna Huang
University of British Columbia
Vancouver, Canada
huicongh@cs.ubc.ca

Caroline Lemieux
University of British Columbia
Vancouver, Canada
clemieux@cs.ubc.ca

Rohan Padhye
Carnegie Mellon University
Pittsburgh, USA
rohanpadhye@cmu.edu

Abstract

Parametric generators are a simple way to combine coverage-guided and generator-based fuzzing. Parametric generators can be thought of as decoders of an arbitrary byte sequence into a structured input. This allows mutations on the byte sequence to map to mutations on the structured input, without requiring the writing of specialized mutators. However, this technique is prone to the *havoc effect*, where small mutations on the byte sequence cause large, destructive mutations to the structured input. This registered report first provides a preliminary investigation of the paradoxical nature of the havoc effect for generator-based fuzzing in Java. In particular, we measure mutation characteristics and confirm the existence of the havoc effect, as well as scenarios where it may be more detrimental. The proposed evaluation extends this investigation over more benchmarks, with the tools Zest, JQF's EI, BeDivFuzz, and Zeugma.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Generator-based Fuzzing, Mutation, Input Generator

1 Introduction

Generator-based fuzzing [15, 27] is a technique for testing programs with randomly generated input data produced via a domain-specific generation function, which samples inputs conforming to some data type or input-format structure. Parametric generators [3, 11, 21, 23,

29–31, 36] enable mutations to be performed on inputs produced by such generators. This unlocks the benefits of coverage-guided grey-box fuzzing [1, 7, 16, 17], which incorporate a feedback loop to guide input generation.

The key idea behind parametric generators is to treat generator functions as *decoders* of an arbitrary sequence of bytes, producing structurally valid inputs given any pseudo-random input sequence. Figure 1 depicts examples of such generators in C++ (via libFuzzer's `FuzzedDataProvider` [8]) and in Java (via JQF [30]) for sampling binary trees; in the latter case, the `Random` parameter is a facade for an object that extracts values from a regular `Input-Stream`. Fig. 2a depicts an example of the decoding process, with bytes color-mapped to corresponding decisions in the generator functions from Fig. 1.

By providing the byte-sequence decoded by the generator to a conventional mutation-based fuzzing algorithm, parametric generators get structured mutations "for free". Fig. 2b shows how a small bit-flip in the byte sequence leads to a small change in the data contained in the corresponding binary tree.

This combination of (a) a method to produce structurally valid inputs, and (b) a method to make small changes to structurally valid inputs, together enables *structure-aware grey-box fuzzing*, resulting in an ability to test deep program states beyond syntax parsing and validation [31]. The high-level insight, popularized by Zest [31], is that *small changes* in the byte-sequence will map to *small changes* in the structured input produced by the generators (e.g., the binary trees)... at least, in theory.

This insight, while compelling, does not always hold. A common criticism of the parametric generator approach is that certain mutations on the byte stream—especially on those bits whose values influence conditional branches in the generator function—can lead to drastic changes in the corresponding structured input being produced. Fig. 2d depicts such a case, where a single bit-flip in the first byte leads to a completely different binary tree being produced; there is almost no similarity to the original tree shown in Fig. 2a. We call this phenomenon the *havoc effect*, inspired by the terminology used by AFL [1] and prior work [38, 41].

Intuitively, the havoc effect appears to be a severe limitation of the parametric generator-based approach to structure-aware grey-box fuzzing because the fuzzing process relies on subtle changes
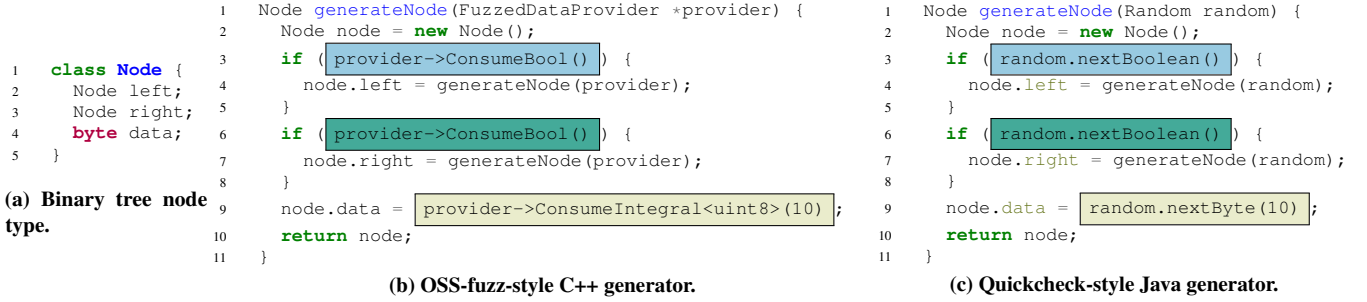
```
1   class Node {
2     Node left;
3     Node right;
4     byte data;
5   }
```

(a) Binary tree node type.

```
1   Node generateNode(FuzzedDataProvider *provider) {
2     Node node = new Node();
3     if ( provider->ConsumeBool() ) {
4       node.left = generateNode(provider);
5     }
6     if ( provider->ConsumeBool() ) {
7       node.right = generateNode(provider);
8     }
9     node.data = provider->ConsumeIntegral<uint8>(10) ;
10    return node;
11  }
```

(b) OSS-fuzz-style C++ generator.

```
1   Node generateNode(Random random) {
2     Node node = new Node();
3     if ( random.nextBoolean() ) {
4       node.left = generateNode(random);
5     }
6     if ( random.nextBoolean() ) {
7       node.right = generateNode(random);
8     }
9     node.data = random.nextByte(10) ;
10    return node;
11  }
```

(c) Quickcheck-style Java generator.

Figure 1: A simplified generator for binary tree nodes in C++ (libFuzzer-style) and Java (JQF-style). While designed for random sampling, grey-box fuzzers such as Zest [31], libFuzzer [7], and AFL [1] supply a deterministic sequence of choices backed by a fixed byte stream that can be mutated.

(a) Seed input

(b) Mutation I.

(c) Mutation II.

(d) Mutation III.
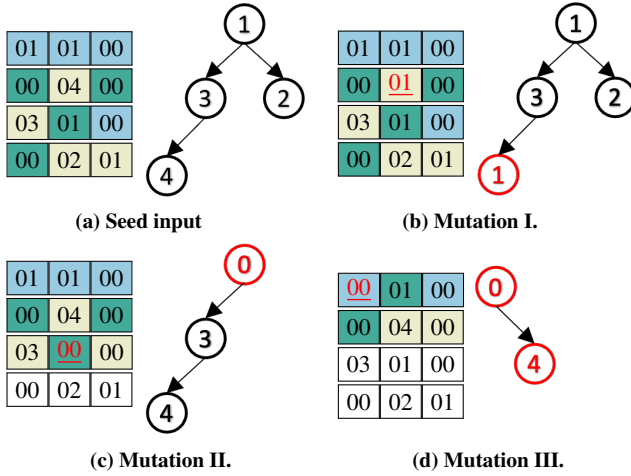
Figure 2: Four inputs as well as their corresponding binary tree object via the `generateNode` method (ref. Fig. 1). The changed bytes are highlighted in <u>red</u>.

to explore program paths incrementally. This unpredictability essentially degrades the effectiveness of grey-box fuzzing, transforming it into a black-box approach where the feedback mechanism is unable to guide the exploration meaningfully due to the havoc effect. Researchers have developed techniques that increase structure-preservation when performing mutations on parametric generators: In JQF [30], the EI backend reduces the destructiveness of mutations by tracking in which generation context the bytes are used; BeDiv-Fuzz [29] separates structure-preserving mutations from structure-changing ones; and Zeugma [21] traces generator execution to enable structure-preserving cross-over across distinct inputs.

We note that structure-aware grey-box fuzzing can be performed without relying on generator functions. The leading alternatives are grammar-based [12, 38, 40] or custom-mutator-based [9, 11] fuzzing, which do not exhibit the havoc phenomenon, but are subject to other trade-offs limiting their expressibility. However, quite paradoxically, at least one set of researchers [38] found that grammar-based grey-box fuzzing actually improves when occasionally using an "aggressive mutation" strategy akin to the havoc effect we described for parametric generators.

The main aim of this registered report is to investigate the paradoxical nature of the havoc effect in generator-based fuzzing by studying the properties of byte-level mutations, their effect on the generated structured inputs, and the performance of generator-based greybox fuzzing under mutation strategies more and less prone to the havoc effect.

In preliminary experiments and investigations, we make three important observations: (i) when using the default strategy of performing random mutations on the underlying bytestream backing parametric generators (as in Zest [31]), the changes to the structured inputs produced by the generator are bimodal—the mutations either change the input very little or by a lot; (ii) the havoc effect can be reduced with context-aware mutations strategies such as JQF's EI; (iii) the havoc effect only destroys the decoding for a suffix of generator execution; therefore, the havoc effect should be benign for programs that process their inputs linearly left-to-right, but may detrimental to programs that pass over the input non-linearly, or multiple times.

As future work, we propose to run a full evaluation that compares the behavior of default parametric generators in Zest [31] with the behavior of JQF's EI, BeDivFuzz [29], and Zeugma [21]. First, we propose to evaluate the characteristics of their mutation, in particular: (1) the observed edit distances over structured inputs, as well as (2) the preservation (or lack thereof) of semantic validity. Second, we propose to evaluate the fuzzing coverage of these four techniques in 24-hour fuzzing runs; and whether their ability to cover program behaviors that go beyond a simple left-to-right pass over the input differs. To address performance discrepancies due to different fuzz framework instrumentation, we also propose a normalized comparative evaluation between these frameworks. Finally, we will evaluate the runtime impact of these techniques, as adding additional context tracking may lead to runtime overhead compared to regular parametric fuzzing.

The main contributions of this registered report are:

- A discussion of the havoc effect in parametric generators, and of its paradoxical impact on saved inputs.
- Preliminary case studies showing that the havoc effect is potentially detrimental to covering program behaviors that go beyond simple a left-to-right pass over the input.
- Preliminary results that show that the havoc effect can be attenuated by smarter mutation strategies.

Algorithm 1: Stream-based `NextByte` method.

```
1  Function NextByte(bytestream B):
2      if HasNext(B) = false then
3          b ← RandomByte()
4          B ← B ∘ b
5      return ReadNext(B)
```

- An evaluation plan to investigate the prevalence of, and overall impact of, the havoc effect in other generator-based fuzzing strategies (Zest, JQF's EI, BeDivFuzz, and Zeguma).

The rest of this paper is structured as follows. Section 2 gives background on the havoc effect, as well as the four generator-based fuzzing strategies we intend to study. Section 3 presents a case study demonstrating that the havoc effect can be detrimental to covering some program behaviors. Section 4 shows the results of our preliminary investigations into quantifying the havoc effect via measurements of mutation distance. Section 5 discusses our evaluation plan, and Section 6 discusses additional related work.

2 Background

We first detail the core technical background for the four techniques we plan to compare: parametric generators, JQF's EI, BeDivFuzz[29] and Zeugma [21].

2.1 Parametric Generators

Parametric generators were introduced by Zest [31] in order to combine Quickcheck [15]-style random generation functions with AFL [1]-style mutation-based grey-box fuzzing. The key idea is to take an off-the-shelf generation function, which queries an API providing pseudo-random values to produce a structurally valid input, and then run that generator with an explicitly provided byte-stream that backs the "random" API. The byte-stream can then be mutated and the generator replayed to get a new structurally valid input.

The data structure `Node` depicted in Figure 1 is a simple example of a structured input. `Node` has three fields: `left`, `right`, and `data`. Figure 1c shows a Quickcheck-style [15] random generator for `Node`. The values returned by the calls to `random.getBoolean()` direct the control-flow through the generator; the values returned by `random.nextByte()` affect the data flow. QuickCheck produces test inputs simply by calling this generator.

This generator can be made *parametric* by "backing" the `random` instance with a given bytestream B. This idea is implemented as follows. The `random.XYZ` methods rely on calls to an internal function `NextByte` that produces pseudo-random bytes. For example, `random.getBoolean()` calls `NextByte` once and returns a boolean value based on whether the result is odd or even. Similarly, `random.nextInt()` calls `NextByte` four times, with the bytes subsequently cast into a four-byte integer. In a regular random generator, `NextByte` uses a source of non-determinism from the operating system to generate the next pseudo-random byte value. In a parametric generator, the implementation of `NextByte` is overridden to provide specific values instead. As detailed in Algorithm 1, `NextByte` takes a bytestream B as input. When invoked, it reads the next byte from B and returns. If B is fully read,

the algorithm generates a new random byte and appends it to B (Line 2). This bytestream-backing of `NextByte`, along with the use of `NextByte` in all `random.XYZ` methods, creates the mapping between *bytestream* and *structured input* in parametric generators.

Figure 2 shows sample bytestreams and their corresponding `Node` object for our running example. Each box represents a byte in hexadecimal notation. The color of the box corresponds to the location where the byte is consumed in Figure 1c.

2.1.1 Havoc Effect The core idea behind the use of parametric generators is that *mutations at the bytestream level* are automatically turned into *mutations at the structured input level*.

Sometimes, these mutations are small. In, Figure 2a fifth byte `04` creates a data value of 4 for the far-left leaf node of the tree. If this fifth byte's value is mutated from `04` to `01`, as in Figure 2b, the structured input is mutated only slightly, with the far-left leaf node taking on the data value 1 instead.

However, mutations can also be much more destructive. In Figure 2a, the first byte (`01`) of the byte stream is consumed by the call to `random.nextBoolean` at Line 3 in Fig. 1c, to decide whether or not to generate a left child for the root node. If this byte's value is mutated from `01` to `00`, as in Figure 2d, we see the generated tree is drastically different from the one in Figure 2a, with a different shape and two fewer nodes. This is because the mutation to the first byte in the bytestream causes all other bytes in the bytestream to be consumed at different locations in the generator, as illustrated by the different colors in the bytestream on the left-hand-side of Figure 2d. Further, the last 6 bytes of the bytestream are not consumed by the generator at all, resulting in the smaller-sized generated tree.

So, while the bytestreams in Figure 2a and Figure 2d are more than 99% similar at the bit-level, the inputs produced by the generator are widely different. We call this tendency for small bytestream mutations to yield large structured input mutations the *havoc effect*.

2.2 Localized Mutations in EI

The JQF framework [30] provides multiple "guidances" or algorithms for driving parametric generators, including Zest (which performs random point mutations on the byte-stream as described above) as well as the structure-preserving `ExecutionIndexingGuidance`, which we refer to as simply EI in this paper.[1]
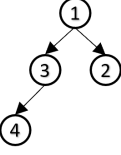
The mutation III in Figure 2d is highly destructive because *the mutation of the first byte causes all subsequent bytes to be consumed at different locations in the generator*. This occurs because `NextByte` processes the bytestream linearly. To make byte-level mutations less destructive, EI uses a representation of the bytestream that associates the *context* in which each byte is consumed.

To represent context, EI uses *execution indexing* [22, 32, 42], which links dynamic execution events across multiple traces [22, 32, 42] (e.g., in Figure 1c, uniquely identifying the "call to `random.nextByte()` setting the data value for the right child of the left child of the root node" across multiple execution paths through the generator during the fuzzing campaign). In EI, each execution index $[(l_1, n_1), ..., (l_i, n_i)]$ uniquely identifies a point in the execution

---

[1]Although the EI implementation in the JQF repository first appears as far back as 2017, there is no published work explaining its logic; so, we provide an expanded description in this paper and for subsequent evaluation use the latest version as of release JQF-2.0 (May 2023).

Table 1: EI-based byte sequence (left) and generated input (right), using the same underlying bytestream as in Figure 2a. The column "Execution Index" lists the context in which each byte is consumed, the column "Data" lists the byte consumed and the corresponding generator "choice".

Execution Index	Data	Execution Index	Data
[(L3, 1)]	01 : T	[(L4, 1), (L9, 1)]	03 : 3
[(L4, 1), (L3, 1)]	01 : T	[(L6, 1)]	01 : T
[(L4, 2), (L3, 1)]	00 : F	[(L7, 1), (L3, 1)]	00 : F
[(L4, 2), (L6, 1)]	00 : F	[(L7, 1), (L6, 1)]	00 : F
[(L4, 2), (L9, 1)]	04 : 4	[(L7, 1), (L9, 1)]	02 : 2
[(L4, 1), (L6, 1)]	00 : F	[(L9, 1)]	01 : 1

Algorithm 2: EI-based `NextByte` and `Mutate` methods.

```
1  Function NextByte(EI-based input M):
2      ei ← CurrentEI()
3      if ei ∈ M then
4          b ← M⟦ei⟧
5      else
6          b ← RandomByte()
7          M[ei ↦ b]
8      end
9      return b
10 Procedure Mutate(EI-based input M):
11     ei ← RandomSelect(M)
12     M[ei ↦ RandomByte()]
13     return M
```

---

trace as a list of tuples analogous to a *call stack*, where each tuple (l_i, n_i) comprises the source location l_i of a method call (i.e., the call site) and the count n_i is an index of how many times l_i has been executed with the context $[(l_1, n_1), ..., (l_{i-1}, n_{i-1})]$.

EI, rather than storing the bytestream backing the pseudo-random as a linear sequence (as in Alg. 1), instead stores the bytestream as a *map* from execution indexes to the byte value consumed at that execution index. Table 1 shows this map given the generator shown in Figure 1c and seed input shown in Figure 2a. The first column shows the execution indexes and the second column presents the associated bytes and its interpreted value. Consider the first row in this table—when the generator consumes the first byte to "choose whether to generate a left child for the root node", the corresponding execution index is [(L3, 1)]. Here, L3 points to Line 3 in the generateNode method containing the call to random.nextBoolean(), and 1 indicates this is the first encounter of this method invocation with nothing else on the call stack. Similarly, the execution index of the third byte (see the third row of Table 1) is [(L4, 2), (L3, 1)]. Here, (L4, 2) indicates that the call stack contains the second call to generateNode at this stack level (i.e., the "left child of the left child of the root node"), where as (L3, 1) indicates that in this context we are considering the first call to nextBoolean() to determine whether to generate another left child. Note that while these bytes are consumed at the same static source code location (i.e., the call to random.nextBoolean at Line 3), they have different dynamic execution indexes, reflecting that their *runtime consumption contexts* are distinct. In fact, execution indexes are unique within a single program execution (i.e., for a single input generation in the fuzzing loop), and so all 12 rows in Table 1 have distinct indexes.

EI then performs structured-input generation and mutation according to the map-based representation M. As shown in Algorithm 2 (contrast with Alg. 1), NextByte does not read the bytestream linearly. Instead, given the current execution index ei where a byte is consumed, NextByte first checks if ei exists in the map M. If it does, NextByte returns the bytes associated with this ei; otherwise, it returns a new random byte and updates M to record the byte consumed. To mutate inputs, as in Mutate in Algorithm 2, EI chooses a random ei in M and mutates the corresponding byte.

This representation enables localized mutations, as seen with a 1-byte mutation in Table 1 that creates a new input in Table 2, compared to the havoc mutation in Figure 2d. In Figure 2d, mutating the first byte of the bytestream—controlling whether the root node

Table 2: EI-based byte sequence (left) and generated input (right) after mutating the 1st byte of the input map depicted in Table 1.

Execution Index	Data	Execution Index	Data
[(L3, 1)]	00 : F	[(L4, 1), (L9, 1)]	03
[(L4, 1), (L3, 1)]	01	[(L6, 1)]	01 : T
[(L4, 2), (L3, 1)]	00	[(L7, 1), (L3, 1)]	00 : F
[(L4, 2), (L6, 1)]	00	[(L7, 1), (L6, 1)]	00 : F
[(L4, 2), (L9, 1)]	04	[(L7, 1), (L9, 1)]	02 : 2
[(L4, 1), (L6, 1)]	00	[(L9, 1)]	01 : 1

should have a left child—caused the generated input to be wildly different. In Table 2, mutating this first byte simply *deleted the left child* without changing the root node or the right child. This is because EI's NextByte, when, e.g., deciding to generate the data of the root node, looks for the bytes consumed at the *same execution index* in the original input, rather than consuming bytes sequentially.

EI otherwise functions similarly to Zest (in terms of deciding which inputs to save, etc.). We hypothesize that EI thus reduces the occurrence of the havoc effect.

One potential limitation of EI is that constructing execution indexes introduces additional overhead of instrumenting the generators, potentially slowing down the fuzzing process. Thus, while EI offers more precise and localized mutations, one trade-off is the increased computational cost.

2.3 Structure Preserving Mutations in BeDivFuzz

Inspired by the parametric generators in Zest, BeDivFuzz is a feedback-driven and generator-based fuzzer that encourages generating valid inputs with behavioral diversity [29]. In particular, BeDivFuzz quantifies the behavioral diversity of input by measuring the effective number of diversely covered branches after executing this input [29]. Driven by this diversity feedback, BeDivFuzz adapts its mutation strategies to generate new inputs.

There are two types of input mutations in BeDivFuzz in addition to the default random mutations as in Zest. The *structure-preserving* mutation allows the testing of a specific behavior of a program with different variants of the same input structure [29]. The *structure-changing* mutation focuses on exploring various program behaviors that are only triggered if the input has a particular structure [29].
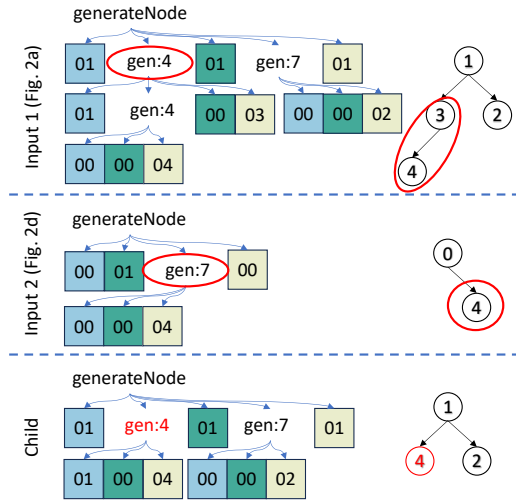
Figure 3: The parametric call tree for the input 1 (Fig. 2a) and input 2 (Fig. 2d). Note that gen:X represents the method call to generateNode at Line X (Fig. 1c). To perform linked-crossover Zeugma slices based on method call boundaries (e.g. the gen method) and creates a new input by replacing the left child node from input 1 with the right child node from input 2.

To achieve these two kinds of mutations, BeDivFuzz splits the random choices in its parametric generator into structural and value choices. Consider the input in Figure 2d as an example. The bytes 1–4 are used to generate boolean values that decide the structure of the binary tree. The bytes 5–6 are used to generate the integer values in the nodes of the binary tree. In this case, bytes 1–4 are *structural* choices and bytes 5-6 are *value* choices. BeDivFuzz requires these choices to be explicitly separated by the generator developer. That is, the blue and teal decision points in Figure 1 (Lines 3, 6) need to labelled as *structural* choices, and the beige decision point (Line 9) needs to be labelled as a *value* choice.

By separating the two types of decision points, and thus, of byte parameters, BeDivFuzz allows for localized mutations to change only the values that do not affect branch conditions while preserving the overall input structures. For the binary tree in Figure 2d, such structure-preserving mutations will only mutate the bytes in the beige cells while keeping the bytes in the blue and teal cells the same. The resulting binary tree will still be a root node with a right child, except that the values in these nodes might change.

2.4 Smart Crossover in Zeugma

Crossover is an effective technique to generate new input by combining parts of multiple inputs together [1, 7, 12, 16, 20, 35]. Traditional crossover techniques, like slicing two inputs at random locations, are ineffective for generated-based fuzzing as they fail to preserve semantic information (e.g., slicing the bytes that generate the left child of the root node). Zeugma addresses this by proposing tree-structured slicing, which slices bytestreams based on method call boundaries (e.g., generateNode in Figure 1) and performs crossover using bytes consumed by the same method.

```
1   void checkLTR(Node n) {      1   void checkRTL(Node n) {
2     if (n.left == null         2     if (n.right == null
3       && n.right == null) {     3       && n.left == null) {
4         // Error path           4         // Error path
5     }                           5     }
6   }                             6   }
```

Figure 4: Code snippets that process input in left-ro-right and in right-to-left-order, relative to the generation order in Fig. 1.

For each saved input, Zeugma generates a *parametric call tree*, decomposing the bytestream into nodes based on method call boundaries. Figure 3 shows the parametric call trees for the inputs in Figure 2a and Figure 2d. In input 1, calling generateNode consumes three bytes and makes two method calls: the call to generateNode at Line 4 (abbreviated as gen:4) to generate the root node's left child, and the call gen:7 for the right child. For linked crossover, Zeugma selects a node in the parent input's call tree that consumes multiple bytes, records the sequence, then finds a node in the supplier input calling the same method and replaces the byte sequence in the parent input with that from the supplier. For instance, gen:4 in input 1 consumes bytes 2-7 to produce the left child, while gen:7 in input 2 consumes bytes 3-5 for the right child. Zeugma can select input 1 as the parent, replacing bytes 2-7 with bytes 3-5 from input 2, resulting in a new binary tree that swaps the left child in input 1 with the right child from input 2 (as shown in Figure 3).

While linked crossover effectively mutates bytestreams while preserving high-level structure, relying solely on it in generator-based fuzzing may limit the generation of interesting inputs. For instance, removing the right child of the root node in the seed input requires changing the 8th byte from 01 to 00, which linked crossover cannot achieve as it only replaces slices. Therefore, Zeugma combines structured linked crossover with existing random byte mutation, as described in Section 2.1.

3 Consequences of the Havoc Effect

It is hard, based on intuition alone, to determine whether the havoc effect is inherently good or bad for fuzzing performance. As aforementioned, Gramatron [38] explicitly added aggressive mutation strategies to improve the performance of grammar-based greybox fuzzing. Similarly, the standard best practice when running AFL variants is to disable the deterministic mutation stage by default [24, 26]. So, in this section, we look at some instances where the havoc effect could have a negative impact on the coverage achieved.

First, consider the small checkLTR and checkRTL functions in Figure 4, which accept as input a single Node, as generated by Figure 1. While the functions are semantically identical, we expect parametric generator-based fuzzing to have more difficulty reaching the error path in checkRTL than checkLTR. This is because the generator in Figure 1 generates the input in a left-to-right order.

The input in Figure 2d, for instance, reaches Line 2 in checkLTR, as it has no left child. To reach the error statement in checkLTR, we need only a single mutation—e.g., to choose whether to generate the right child. This is because the left child is generated first, and so a mutation *after* the left child is chosen to be not generated will not affect the left child.

On the other hand, suppose the fuzzer first performs mutation II shown in Figure 2c, which generates a tree whose root node has

no right child. This input reaches Line 2 in `checkRTL`. However, any single mutation to the bytestream that affects the choice to generate the left child would change how all the subsequent bytes are interpreted, similarly to mutation III in Figure 2d. Thus, no single-byte mutation allows the fuzzer to remove the left child, which preserves the absence of the right child. We use EI and Zest to analyze `checkLTR` and `checkRTL`, allowing each fuzzer one million trials, repeated 1000 times. Both EI and Zest achieve a 100% error path discovery rate for `checkLTR`, meaning they consistently trigger the error path in all 1000 repetitions. For `checkRTL`, EI maintains a 100% error path discovery rate, while Zest achieves this in only 79% of the repetitions.

The natural question is whether such input reading (e.g., reading the input in non-generated order or reading the input multiple times) behavior occurs in the wild. To investigate this, we ran some preliminary experiments on the closure benchmark. Closure is a Javascript compiler written in Java, used in the original benchmark suite for Zest [31]. In particular, we ran both Zest [31] and JQF's EI on this benchmark. Examining several 24-hour runs, we saw JQF's EI had the ability to cover one particular code fragment, which Zest did not.

This code fragment is illustrated in Figure 5. In particular, Figure 5 shows the process by which EI identifies an input covering a branch in the closure compiler's cost estimator for all our initial experiments. This branch is not covered by Zest. A seed input (top left) covers four statements within the `costEstimator` method. In the seed input, the statement `call(foo)` is a method call with a function pointer as an argument. This call triggers the cost estimation path for the `foo` method. To estimate the execution cost for `foo`, the estimator proceeds by iterating and analyzing the cost of each statement enclosed in `foo`. Since the definition of `foo` in the seed input includes a for statement, the seed input covers the `analyzeFor` statement found within the `costEstimator` method.

We note that the generator used in these experiments generates the JavaScript program in sequential order. Initially, it constructs the `foo` method. Following this, it generates the `call(foo)` statement. However, the `costEstimator` method processes the input in reversed order. It interprets the statements contained within the `foo` method if and only if the `call(foo)` expression exists.

To cover the `analyzeIf` statement from here, a mutated input needs to add an if statement into the `foo` method while keeping the `call(foo)` statement. Let's consider a scenario where both Zest and EI successfully generate a mutation that introduces an if statement after the for statement. Thanks to the structure-preserving mutations provided by execution indexing, EI is able to insert this if statement without disturbing the subsequent statements (top middle of Figure 5). Conversely, due to the havoc effect, this mutation in Zest removes the `call(foo)` statement (bottom middle of Figure 5). When the closure compiler is executed with these mutated inputs, the one generated by EI successfully extends the coverage to include the `analyzeIf` statement. In contrast, the Zest-generated input fails to augment the coverage: the `call(foo)` statement has been turned into a return statement, so the code guarded by `methodCallWithFunction(input)` is not executed.

This example motivates the value of reducing the havoc effect in order to cover certain program behaviors. Thus, we propose to examine whether different generator-based coverage-guided fuzzing systems (Zest, EI, BeDivFuzz, Zeugma) demonstrate the havoc effect, as well as whether this results in an overall positive or negative effect on fuzzing performance on a broader benchmark set.

Table 3: Characteristics of our benchmark applications.

Benchmark	Generator	LOC	Gen-LOC
Chocopy [33]	Python	6K	397
Gson [5]	Json	26K	89
Jackson [6]	Json	49K	89
Closure Compiler [4]	JavaScript	250K	250
Rhino Compiler [10]	JavaScript	110K	250
Maven [28]	XML	93K	136
Ant [2]	XML	140k	136

4 Preliminary Results

The first goal of our preliminary evaluation is to investigate if the havoc effect is present and measurable in Zest. We then also investigate whether the structure-preserving mutations in EI indeed reduce the havoc effect in parametric generators. Finally, we look at some preliminary coverage measurements on our selected fuzzers.

Benchmarks We selected 7 different benchmark programs with 4 different program generators used by prior research [31, 39]. Table 3 shows the detailed characteristics, including the benchmark name, the generator used to generate input data, the lines of code of the benchmark program, and the lines of code of the input generator.

4.1 Measuring the Havoc Effect in Zest

To understand how destructive the mutations are in Zest, we measure how different mutated inputs are from their parents. To evaluate this, we choose to compute the normalized Levenshtein distance between a mutated input and its parent. We call this *mutation distance*. When the mutant and its parent are the same, the normalized mutation distance is 0. The greater the distance is, the more different the mutant is from its parent. In our measurements, we noticed a number of inputs with 0 mutation distance. We exclude these mutants from our analysis as they are not mutants and could trivially be filtered out by the fuzzer; we discuss this issue further in Section 4.2.1.

Figure 6 compares the distribution of normalized mutation distances for all inputs generated by Zest (left, orange) and the inputs saved to the fuzzing corpus (right, lavender). This allows us to compare the distribution of all generator-produced inputs to those inputs that are useful for the fuzzing process (i.e., the saved inputs).

We observe significant variations in the magnitude of mutation distances across different benchmarks. However, benchmarks with the same generator have similar distributions, underscoring the impact of the generator's implementation on the havoc effect.

For simple targets like gson and jackson, which parse JSON strings into Java objects, the havoc effect appears to be beneficial, as a higher frequency of saved inputs has large mutation distances. Therefore, we do not expect structure-preserving mutations to improve the fuzzing performance for these benchmarks significantly.

The mutation distance distribution of all inputs for ant and maven, closely mirrors that of the saved inputs, suggesting that parametric generators do a good job of generating mutants for these benchmarks. Unlike the other benchmarks, we see no peak in the Kernel Density Estimate (KDE) curves at high mutation distances, suggesting the havoc effect is not as present for the XML generator these
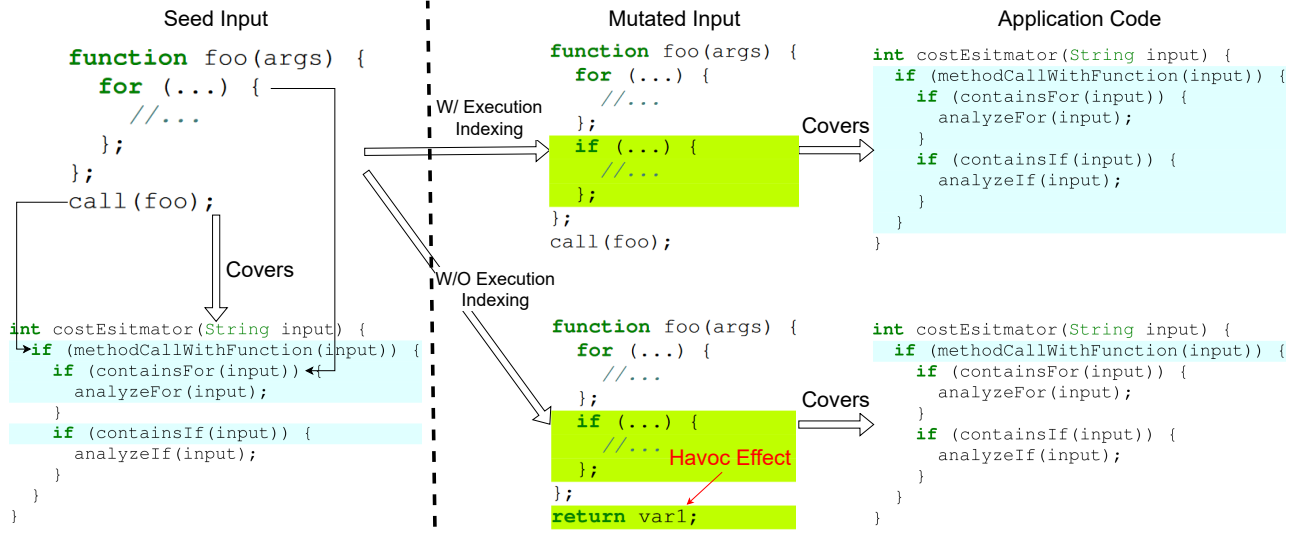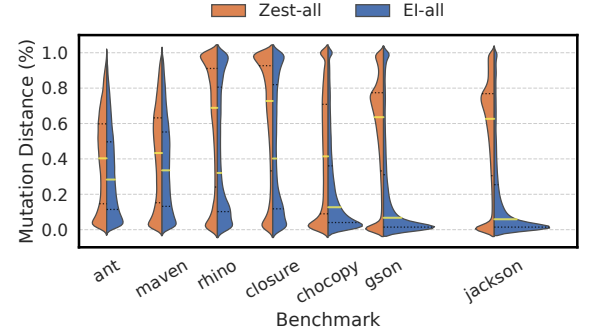
Figure 5: Case study of fuzzing the Closure Javascript compiler (written in Java). EI can find an input that covers the `analyzeIf` statement, which requires introducing an `if` statement before the statement calling `foo`; this is hard for Zest to cover due to the havoc effect which disrupts the suffix of the input containing the `foo` call whenever Zest manages to generate an `if` statement.
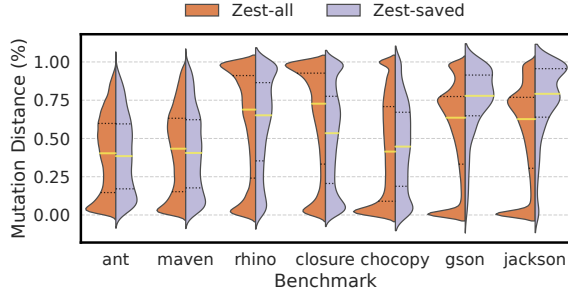
Figure 6: Mutation distance distributions for all inputs (left, orange half) and saved inputs (right, lavender half) generated by Zest. A wider area means a higher frequency of that mutation distance. The yellow line highlights the median mutation distance, and the dotted lines are the quartiles.

Figure 7: Distribution of mutation distances for all inputs generated by Zest and EI. EI generates inputs with small mutation distances at higher frequencies for all benchmarks.

benchmarks share. In addition, the mutation distance for all inputs exhibits a denser cluster in the lower range, further suggesting that structure-preserving mutations may not affect these benchmarks.

For more complex targets such as rhino and closure, the frequency curves for all inputs exhibit two distinct peaks—one at extremely small mutation distances and another at large mutation distances. This dual-peaked pattern shows that rhino and closure are affected by the havoc effect as the small byte-level mutations in Zest result in many inputs with large mutation distances. For closure, the median mutation distance for saved inputs is significantly lower than that for all inputs. The KDE curve for saved inputs in closure exhibits a higher peak at smaller mutation distances, showing a greater density of produced inputs with smaller mutations. Thus, closure may benefit more the structure-preserving mutations.

4.2 Can We Reduce the Havoc Effect?

Is the havoc effect inherent to all generator-based greybox fuzzers, or does it differ? We conduct a preliminary evaluation of this by comparing the mutation distances of a Zest to EI, which should suffer less from the havoc effect. In Figure 7, we plot the mutation distance distribution for all inputs generated by Zest and EI. Figure 7 shows that the median mutation distances for all inputs generated by EI are much lower than Zest. This is in line with our expectations, where EI can more effectively localize mutations. Notably, for chocopy, gson, and jackson, the EI part of the violins peak at the bottom, demonstrating a denser cluster of inputs with small mutation distances. The inputs generated by EI for ant and maven have a similarly high frequency of small mutation distances. Though EI's mutation distance distributions for rhino and closure are bimodal, the KDE curves have a higher peak at the smaller mutation distances. In contrast, the KDE curves for Zest do not always reach the highest peak at the bottom and have multiple modes across a wide range
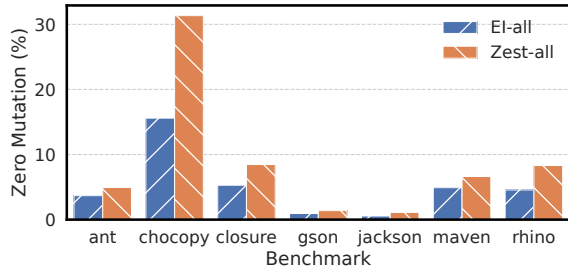
Figure 8: Percentage of zero mutations (i.e., mutants identical to parent) out of all generated inputs for EI and Zest.

Table 4: The average branch coverage and execution count for each fuzzer (rounded to the nearest thousand) in application classes for Closure across 20 fuzzing campaigns after 24 hours.

	Zest	EI	Zeugma	BDF(S)	BDF(D)
Branch Coverage	24,203	24,252	24,990	23,809	23,574
# Executions ($\times 10^3$)	5,542	5,555	10,413	4,687	5,009

of mutation distances. Our findings confirm that the EI-based approach indeed alleviates the destructiveness of mutations witnessed in stream-based approaches such as Zest. More importantly, this finding is consistent across different benchmarks and generators.

4.2.1 Zero Mutations As aforementioned, an unexpected observation gleaned during our preliminary evaluation was the frequent occurrence of inputs with 0 mutation distance. We refer to these mutations where the mutated input is identical to the parent input as the *zero mutations*. As discussed in Section 4.1, we removed zero mutations from our earlier analysis: a duplicate input will never be saved and should ideally be filtered out by the fuzzing algorithm.

Zero mutations occur when the fuzzer's mutation to the input bytestream does not change the generator-produced input, effectively creating a duplicate input. For example, in Figure 2a, where altering the first byte to either `00` or `02` results in identical binary trees, because both mappings yield false in the generator (Line 3). Figure 8 shows the frequency of zero mutations in inputs generated by Zest and EI. We see that EI generates a much smaller proportion of zero mutations than Zest. This should lead to better fuzzer efficiency. In future work, we will also study the occurrence of zero mutations in BeDivFuzz and Zeugma.

4.3 Preliminary Coverage Comparison

To assess the impact of destructive mutations on code coverage, we selected four fuzzing techniques: Zest [31], EI, BeDivFuzz [29], and Zeugma [21]. For BeDivFuzz, we consider two configurations used by the prior research [21, 29]: BDF(S), which only includes structural mutation, and BDF(D), which includes both structural mutation and feedback of input structure novelty. Each technique was used to fuzz Closure for 24 hours, repeated 20 times. Since Zeugma is implemented in a different fuzzing framework, we also recorded the number of total executions to understand potential alternative causes for variations in branch coverage, such as differences in fuzzing speed.

Table 4 shows the average branch coverage for each technique. Although EI achieves higher average coverage than Zest, the Mann-Whitney U test result shows no significant difference. BeDivFuzz shows lower executions compared to Zest and EI, with significantly lower branch coverage. Zeugma significantly outperforms Zest and EI in branch coverage, but the number of executions achieved by Zeugma is also significantly higher (almost 2×)—we posit that at least some of the speed is due to engineering improvements, since Zeugma is the only tool from the ones we studied that is not built on top of JQF but instead implemented afresh from the ground up. Thus, we are not certain whether Zeugma's higher branch coverage in a fixed time budget is attainable solely to the linked-crossover technique or the difference in execution speed. We plan to conduct further experiments to isolate and control for this difference.

5 Proposed Evaluation

Our proposed evaluation aims to answer the following questions:
RQ1: How destructive, in terms of edit distance, are the mutations performed by our studied techniques?
RQ2: How much do the mutations performed by our studied techniques preserve semantic validity?
RQ3: How does the achieved coverage of our studied techniques compare, especially with longer timeouts?
RQ4: Do the more complex mutations of our studied techniques incur runtime overhead?

In order to answer these questions, we will follow the following experimental procedures.
Expanded evaluation with more targets and benchmarks. We plan to broaden our evaluation to assess the impact of the havoc effect in generator-based fuzzers on their performance. First, we will measure the mutation distance for Zeugma and BeDivFuzz on all targets listed in Table 3. Our initial evaluation indicates that EI consistently mitigates the havoc effect across all benchmark applications. We aim to extend this analysis to Zeugma and BeDivFuzz to determine if their structure-preserving mutations similarly reduce the havoc effect. We would also like to understand how zero mutations manifest in Zeugma and BeDivFuzz.

Second, we would like to understand the impact of structure-preserving mutations on semantic validity. A key advantage of parametric generator-based fuzzing is its capability to generate inputs towards semantic validity. In the proposed evaluation, we would like to understand if the structure-preserving mutations enhance the generation of semantically valid inputs. Following prior work [31], we will use the semantic validity signals (e.g., type-checking result in Chocopy) from each of the targets (Table 3) to determine if a given input is deemed valid or not.

Finally, we will extend our coverage measurement (ref. Section 4.3) to all benchmarks. Our goal is to study the relationship between the magnitude of the havoc effect and the branch coverage across all benchmarks using all of the studied fuzzing tools.
Normalized comparative evaluation between frameworks. One issue in our preliminary results is that different tools are built on top of different fuzzing frameworks. For example, Zeugma uses lighter instrumentation strategy, achieving higher fuzzing speed compared to Zest, EI, and BeDivFuzz (which are implemented on top of JQF, thus paying a performance penalty for the extensibility of the underlying framework). To ensure a fair comparison of branch coverage without

reimplementing all techniques in a single framework, we propose a normalized comparative evaluation. Specifically, we will compare the execution speeds of Zest and Zeugma-without-linked-crossover (Zeugma-X in the original paper [21]) to isolate the performance improvements introduced by the Zeugma framework without introducing a new mutation technique. The difference between these two configurations will give us an adjustment factor to compensate for the difference in the underlying instrumentation framework, which we will then apply to artificially slow down Zeugma (by reducing its total running time) when comparing with the JQF-based techniques for coverage measurement. Note that while this slowing down Zeugma appears "unfair", recall that our goal is not to identify the best fuzzing tool for a practitioner to use, but instead to study the nuances of the havoc effect on coverage—the handicap should enable this study without bias.

6 Additional Related Work

Grammar-based Input Generation. There is a rich body of work focusing on grammar-based input generation for greybox fuzzing [9, 12, 13, 19, 37, 38, 43]. Gramartron translates context-free grammar into a grammar automaton, simultaneously introducing havoc mutations as a strategy to overcome the limitations of small-scale mutations, which it argues can inefficiently consume a fuzzer's time by focusing on localized, minor changes [38]. However, the effectiveness of such a mutation strategy has been questioned in subsequent research [17], which suggests that havoc mutations may not consistently yield improvements in generator-based coverage-guided fuzzing. Our work builds on this discourse by being the first to conduct a comprehensive analysis of the havoc effect within the realm of generator-based fuzzing, aiming to understand how havoc effect affects the performance of coverage-guided fuzzing.

Improve mutation precision. The pursuit of enhanced mutation precision has led researchers to develop various techniques, predominantly in two areas: (1) leveraging input grammar to refine mutation algorithms, exemplified by tools like DIE [34] and Tzer [25]; and (2) employing dynamic analysis to pinpoint bytes of interest for mutation, as seen in approaches such as Confetti [23], Grey-One [18], and Angora [14]. Specifically, DIE [34] enhances mutation effectiveness by preserving "interesting" types and structures, which focuses the search process. In a similar vein, Tzer [25] integrates tensor-compiler-specific mutations—tailored for deep learning systems—with general-purpose mutation strategies to achieve a balanced mix of exploration and exploitation. In contrast to these domain-focused methods, the EI-based generator distinguishes itself with its versatility. It is uniquely adaptable across various generator-based fuzzers without relying on domain-specific knowledge, thereby broadening its applicability.

7 Revision Requirements

In the next revision, we will address the issues raised by the reviewers. Firstly, we will clearly state our expectations for the final evaluation. Secondly, we will add comparing the fuzzers based on the number of executions in addition to time to our proposed evaluation, and clarify the reason for running longer fuzzing campaigns. Lastly, we will simplify Sections 2 and 3 to improve readability.

References

[1] [n. d.]. American Fuzzy Lop. https://github.com/google/AFL. Accessed: 2021-08-31.
[2] [n. d.]. Apache Ant is a Java-based build tool. https://github.com/apache/ant.
[3] [n. d.]. cargo-fuzz. https://github.com/rust-fuzz/cargo-fuzz. Accessed: 2023-05-01.
[4] [n. d.]. Closure Compiler. https://developers.google.com/closure/compiler. Accessed: 2021-08-31.
[5] [n. d.]. GSON: A Java serialization/deserialization library to convert Java Objects into JSON and back. https://github.com/google/gson.
[6] [n. d.]. Jackson Project Home @github. https://github.com/FasterXML/jackson.
[7] [n. d.]. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. Accessed: 2021-08-31.
[8] [n. d.]. libFuzzer – How To Split A Fuzzer-Generated Input Into Several Parts. https://github.com/google/fuzzing/blob/41d7725/docs/split-inputs.md. Accessed: 2021-08-31.
[9] [n. d.]. libprotobuf-mutator. https://github.com/google/libprotobuf-mutator. Accessed: 2021-08-31.
[10] [n. d.]. Rhino: JavaScript in Java. https://github.com/mozilla/rhino.
[11] [n. d.]. Structure-Aware Fuzzing with libFuzzer. https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md. Accessed: 2022-06-02.
[12] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars.. In *NDSS*.
[13] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. 2007. Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '07)*. Association for Computing Machinery, New York, NY, USA, 317–329. https://doi.org/10.1145/1315245.1315286
[14] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.
[15] 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*. Association for Computing Machinery, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266
[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.
[17] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. 2022. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1051–1065. https://doi.org/10.1145/3548606.3560602
[18] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2577–2594.
[19] Harrison Green and Thanassis Avgerinos. 2022. GraphFuzz: Library API Fuzzing with Lifetime-Aware Dataflow Graphs. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1070–1081. https://doi.org/10.1145/3510003.3510228
[20] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *21st USENIX Security Symposium (USENIX Security 12)*. USENIX Association, Bellevue, WA, 445–458. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler
[21] Katherine Hough and Jonathan Bell. 2024. Crossover in Parametric Fuzzing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.
[22] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. 2009. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 110–120. https://doi.org/10.1145/1542476.1542489

[23] James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. 2022. Confetti: Amplifying concolic guidance for fuzzers. In *Proceedings of the 44th International Conference on Software Engineering*. 438–450.

[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 International Conference on Automated Software Engineering*. 475–485.

[25] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. 2022. Coverage-Guided Tensor Compiler Fuzzing with Joint IR-Pass Mutation. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 73 (apr 2022), 26 pages. https://doi.org/10.1145/3527317

[26] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1393–1403. https://doi.org/10.1145/3468264.3473932

[27] Charlie Miller, Zachary NJ Peterson, et al. 2007. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep* 4 (2007).

[28] Frederic P Miller, Agnes F Vandome, and John McBrewster. 2010. *Apache Maven*. Alpha Press.

[29] Hoang Lam Nguyen and Lars Grunske. 2022. BEDIVFUZZ: Integrating Behavioral Diversity into Generator-based Fuzzing. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 249–261. https://doi.org/10.1145/3510003.3510182

[30] 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* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 398–401. https://doi.org/10.1145/3293882.3339002

[31] 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*. 329–340.

[32] R. Padhye and K. Sen. 2017. Travioli: A Dynamic Analysis for Detecting Data-Structure Traversals. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 473–483. https://doi.org/10.1109/ICSE.2017.50

[33] Rohan Padhye, Koushik Sen, and Paul N. Hilfinger. 2019. ChocoPy: A Programming Language for Compilers Courses. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E* (Athens, Greece) *(SPLASH-E 2019)*. Association for Computing Machinery, New York, NY, USA, 41–45. https://doi.org/10.1145/3358711.3361627

[34] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1629–1642. https://doi.org/10.1109/SP40000.2020.00067

[35] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering* (2019).

[36] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. 2020. Quickly generating diverse valid test inputs with reinforcement learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1410–1421.

[37] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wör-ner, and Thorsten Holz. 2021. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2597–2614. https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo

[38] Prashast Srivastava and Mathias Payer. 2021. Gramatron: Effective Grammar-Aware Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 244–256. https://doi.org/10.1145/3460319.3464814

[39] Vasudev Vikram, Isabella Laybourn, Ao Li, Nicole Nair, Kelton OBrien, Rafaello Sanna, and Rohan Padhye. 2023. Guiding Greybox Fuzzing with Mutation Testing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (, Seattle, WA, USA,) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 929–941. https://doi.org/10.1145/3597926.3598107

[40] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 724–735.

[41] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. 2022. One Fuzzing Strategy to Rule Them All. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1634–1645. https://doi.org/10.1145/3510003.3510174

[42] Bin Xin, William N. Sumner, and Xiangyu Zhang. 2008. Efficient Program Execution Indexing. 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, 238–248. https://doi.org/10.1145/1375581.1375611

[43] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 955–970. https://doi.org/10.1145/3372297.3417260