The Havoc Paradox in Generator-Based Fuzzing

AO LI*, Carnegie Mellon University, USA
MADONNA HUANG*, University of British Columbia, Canada
VASUDEV VIKRAM*, Carnegie Mellon University, USA
CAROLINE LEMIEUX, University of British Columbia, Canada
ROHAN PADHYE, Carnegie Mellon University, USA

Parametric generators combine coverage-guided and generator-based fuzzing for testing programs requiring structured inputs. They function as decoders that transform arbitrary byte sequences into structured inputs, allowing mutations on byte sequences to map directly to mutations on structured inputs, without requiring 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 paper investigates 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. Our evaluation across 7 real-world Java applications compares various techniques that perform context-aware, finer-grained mutations on parametric byte sequences, such as JQF-EI, BeDivFuzz, and Zeugma. We find that these techniques exhibit better control over input mutations and consistently reduce the havoc effect compared to our coverage-guided fuzzer baseline Zest. While we find that context-aware mutation approaches can achieve significantly higher code coverage, we see that destructive mutations still play a valuable role in discovering inputs that increase code coverage. Specialized mutation strategies, while effective, impose substantial computational overhead—revealing practical trade-offs in mitigating the havoc effect.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Generator-based Fuzzing, Mutation, Input Generator

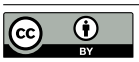
ACM Reference Format:
Ao Li, Madonna Huang, Vasudev Vikram, Caroline Lemieux, and Rohan Padhye. 2025. The Havoc Paradox in Generator-Based Fuzzing. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2025), 26 pages. https://doi.org/10.1145/3742894

1 Introduction

Generator-based fuzzing [19, 36] 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, 26, 29, 38–40, 45] enable mutations to be performed on inputs produced by such generators. This unlocks the benefits of coverage-guided grey-box fuzzing [1, 7, 20, 21], which incorporates a feedback loop to guide input generation.

*All student authors made significant contributions to the paper.

Authors' Contact Information: Ao Li, Carnegie Mellon University, Pittsburgh, USA, aoli@cmu.edu; Madonna Huang, University of British Columbia, Vancouver, Canada, madonna.huang@gmail.com; Vasudev Vikram, Carnegie Mellon University, Pittsburgh, PA, USA, vasumv@cmu.edu; Caroline Lemieux, University of British Columbia, Vancouver, Canada, clemieux@cs.ubc.ca; Rohan Padhye, Carnegie Mellon University, Pittsburgh, USA, rohanpadhye@cmu.edu.



ACM 1557-7392/2025/1-ART1
https://doi.org/10.1145/3742894

```
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.

Fig. 1. A simplified generator for binary tree nodes in C++, Fig. 1. (b) libFuzzer-style, and Java, Fig. 1. (c) JQF-style. While designed for random sampling, grey-box fuzzers such as Zest [40], libFuzzer [7], and AFL [1] supply a deterministic sequence of choices to generate certain input structures. These choices are made from a fixed byte stream, which can be mutated.

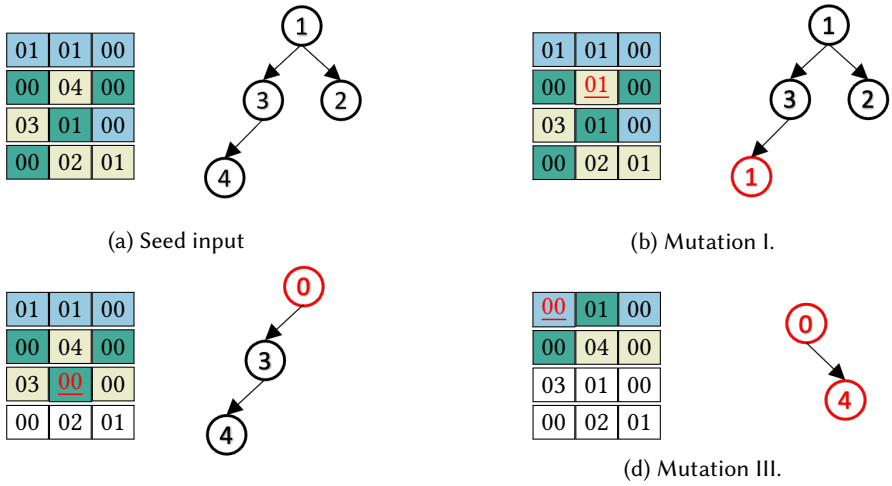


Fig. 2. Four inputs as well as their corresponding binary tree object via the `generateNode` method (ref. Fig. 1). The changed bytes are highlighted in red.

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. Fig. 1 depicts examples of such generators in C++ (via libFuzzer's `FuzzedDataProvider` [8]) and in Java (via JQF [39]) for sampling binary trees; in the latter case, the `Random` parameter is a facade for an object that extracts values from a regular `InputStream`. Fig. 2a depicts an example of the decoding process. The illustrated bytes are color-mapped to decisions that consume them 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 single byte mutation 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*. This results in an ability to test deep program states beyond syntax parsing and validation [40]. The

high-level insight, popularized by Zest [40], 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 between Fig. 2d and the original tree in Fig. 2a. We call this phenomenon the *havoc effect*, inspired by the terminology used by AFL [1] and prior work [48, 53]. Havoc in AFL is a mutation stage where multiple mutations are stacked to create new inputs, which tend to differ substantially from parent inputs. Similarly, the havoc effect refers to cases where newly-created inputs differ substantially from their parent inputs.

Intuitively, the havoc effect appears to be a severe limitation of the parametric generator-based approach to structure-aware grey-box fuzzing. Grey-box fuzzing relies on subtle changes to explore program paths incrementally. If mutants are too far from their parents, grey-box fuzzing degrades into a black-box approach, where the feedback mechanism is unable to guide the exploration. We thus expect that unpredictability from the havoc effect will make grey-box fuzzing less effective.

Because of this, researchers have developed techniques that perform *context-aware* mutations that aim to better preserve structure of the original input. In JQF [39], the EI backend reduces the destructiveness of mutations by tracking in which generation context the bytes are used. BeDivFuzz [38] separates structure-preserving mutations from structure-changing ones. Zeugma [26] 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 [13, 48, 51] 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 [48] 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 paper 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 our investigation of the havoc effect, we conducted a thorough evaluation of generator-based fuzzing techniques on 7 real-world Java applications. We introduce *mutation distance*, i.e. the Levenshtein distance between the parent and child input, in order to better quantify the havoc effect. Our analysis reveals five key insights: (i) context-aware mutation strategies such as JQF's EI, BeDivFuzz, and Zeugma significantly reduce the havoc effect—demonstrated by how closely the byte-level changes correspond to the resulting string-level changes in the generated test inputs—as compared to baseline techniques; (ii) context-aware mutations better preserve input validity than random and Zest; (iii) techniques employing context-aware mutations (EI, Zeugma) can achieve significantly higher code coverage than Zest, with EI showing up to 2.6% coverage improvement on certain benchmarks and Zeugma achieving up to 8.3% higher coverage on others; (iv) despite the benefits of structure preserving mutations, more destructive mutations remain valuable for code exploration, as evidenced by our finding that mutations producing coverage-increasing inputs typically have 1.2×–1.4× higher mutation distances than average for BeDivFuzz and EI; (v) techniques using context-aware strategies incur runtime overhead ranging from 15-80% depending

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)
```

on the benchmark and technique, presenting a clear trade-off between structured input exploration and execution speed.

The main contributions of this paper are:

- A discussion of the havoc effect in parametric generators, and of its paradoxical impact on saved inputs.
- Case studies demonstrating that the havoc effect negatively impacts coverage of program behaviors that require more than simple left-to-right parsing, particularly affecting complex semantic features in structured inputs.
- A comprehensive evaluation across seven real-world Java benchmarks that compares how state-of-the-art generator-based fuzzing strategies (Zest, JQF's EI, BeDivFuzz, and Zeugma) mitigate the havoc effect. We introduce the concept of mutation distance to measure the havoc effect in each of these techniques. The artifact is available at https://github.com/cmu-pasta/havoc-mutation-eval.
- Empirical evidence showing that less mutation destructiveness can result in higher code coverage, providing practical insights for future generator-based fuzzing algorithm design.

2 The Havoc Effect and its Potential Mitigation

We first detail the core technical background for parametric generators in order to understand why the havoc effect occurs. Then, we describe EI as an approach to mitigate the havoc effect.

2.1 Parametric Generators

Parametric generators were introduced by Zest [40] in order to combine Quickcheck [19]-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 Fig. 1 is a simple example of a structured input. `Node` has three fields: `left`, `right`, and `data`. Fig. 1c shows a Quickcheck-style [19] Java 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. A QuickCheck-like testing system (e.g., junit-quickcheck [25]) can produce 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 Alg. 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 2-4). 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.

Fig. 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 Fig. 1c.

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 Fig. 2a, the 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 Fig. 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 Fig. 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 Fig. 2d, we see the generated tree is drastically different from the one in Fig. 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 Fig. 2d. Further, the last 6 bytes of the bytestream are not consumed by the generator at all, resulting in the smaller-sized tree.

So, while the bytestreams in Fig. 2a and Fig. 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 with Execution Indexing

The JQF framework [39] 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 Fig. 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* [27, 41, 54], which links dynamic execution events across multiple traces (e.g., in Fig. 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 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 $[(l_1, n_1), ..., (l_{i-1}, n_{i-1})]$.

[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).

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

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

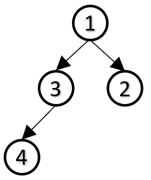


Fig. 3. EI-based byte sequence (left) and generated input (right), using the same underlying bytestream as in Fig. 2a. The column "Execution Index" lists the context in which each byte is consumed, the column "Data" lists the byte consumed and the corresponding generator "choice".

EI, rather than storing the bytestream backing the pseudo-random instance 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. Fig. 3 shows this map given the generator shown in Fig. 1c and seed input shown in Fig. 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 Fig. 3) 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 Fig. 3 have distinct indexes.

EI then performs structured-input generation and mutation according to the map-based representation M. As shown in Alg. 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 Alg. 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 Fig. 3 that creates a new input in Fig. 4. In contrast, the havoc mutation shown in Fig. 2d demonstrates how altering the first byte of the bytestream—controlling whether the root node should have a left child—leads not only to the removal of the left child but also to the creation of a right child. In Fig. 4, 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 generators, potentially slowing down the fuzzing process. Thus, while

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
```

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

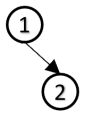


Fig. 4. EI-based byte sequence (left) and generated input (right) after mutating the 1st byte of the input map depicted in Fig. 3. The mutated byte is highlighted in red.

EI offers more precise and localized mutations, it may increase computational cost, resulting in lower fuzzing throughput.

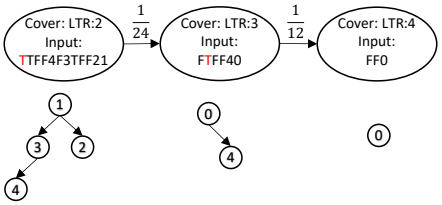
3 Consequences of the Havoc Effect

It is hard, based on intuition alone, to determine whether the havoc effect is inherently good or bad for fuzzing performance. As aforementioned, Gramatron [48] 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 [33, 35]. 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 Fig. 5, which accept as input a single `Node`, as generated by Fig. 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 Fig. 1 generates the input in a left-to-right order.

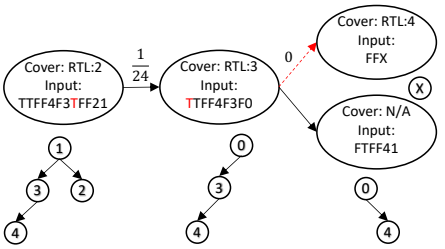
We illustrate this with a Markov chain model (similar to AFLFast [14]). For simplicity, let us assume the fuzzer mutates only one byte at a time. The seed input (Fig. 2a) covers Line 2 in both `checkLTR` and `checkRTL`. But, as neither of the conditions in Line 2 are satisfied, evaluation short-circuits, and the seed input does not cover Line 3 in either `checkLTR` or `checkRTL`.

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



(a) Left-to-right example.

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



(b) Right-to-left example.

Fig. 5. Code snippets that process input in left-to-right and in right-to-left-order, relative to the generation order in Fig. 1. The Markov chain model represents the probability of the fuzzer discovering a new input that covers a new branch, given a seed input. Each node indicates the lines covered by the input and the sequence of choices made by the generator. The byte highlighted in red represents the one that must be mutated to produce the next input. Edges indicate saved mutations, with the numbers representing the probability of the fuzzer performing the corresponding mutation.

The Markov chain model in Fig. 5a illustrates the process by which the fuzzer covers the error path in checkLTR. Initially, the fuzzer mutates the first byte of the seed input, resulting in the removal of the root's left child, as depicted in Fig. 2d. This mutation allows the new input to reach Line 3, and the fuzzer saves this input for further mutations. Subsequently, the fuzzer mutates the second byte, removing the root's right child. The mutated input thus satisfies the condition at Line 3, leading to the coverage of the error path.

On the other hand, given the same input, the fuzzer cannot cover the error branch in CheckRTL no matter how many mutations are performed, as shown in Fig. 5b. Suppose the fuzzer first performs mutation II shown in Fig. 2c, which generates a tree whose root node has no right child. This input is saved by the fuzzer because it reaches Line 3 in checkRTL, but the condition is not satisfied. In order to satisfy the condition at Line 3, the following mutation needs to remote the left child. However, any single mutation in 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 Fig. 2d. Thus, no single-byte mutation allows the fuzzer to remove the left child, which preserves the absence of the right child (the red dashed line in Fig. 5b), allowing both conditions at Line 3 and Line 2 to be satisfied.

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 Google Closure benchmark. Closure is a Javascript compiler written in Java, used in the original benchmark suite for Zest [40]. In particular, we ran both Zest [40] 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.

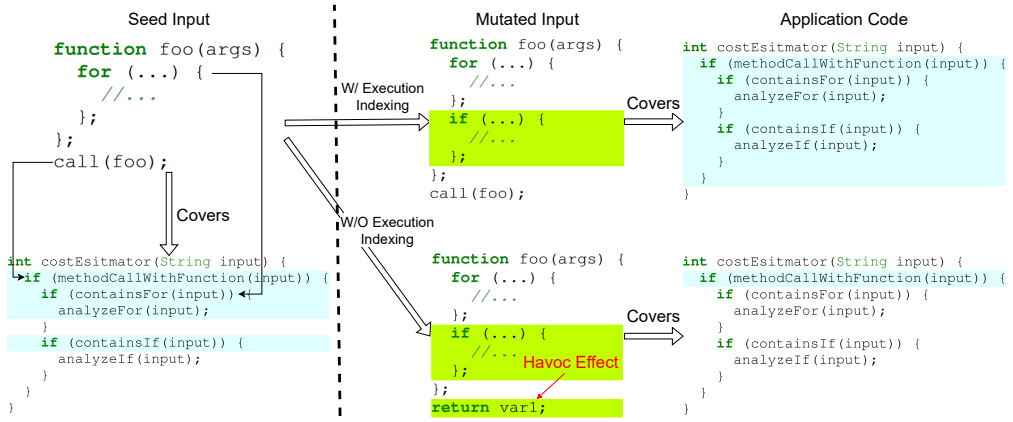


Fig. 6. Case study of fuzzing the Closure Javascript compiler (written in Java). EI can find an input that covers the `analyzeIf` statement, which requires introducing an `if` statement before the statement calling `foo`; this is hard for Zest to cover due to the havoc effect which disrupts the suffix of the input containing the `foo` call whenever Zest manages to generate an `if` statement.

This code fragment is illustrated in Fig. 6. Fig. 6 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 (highlighted in cyan in Fig. 6) 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.

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 its structure-preserving mutations, EI is able to insert this if statement without disturbing the subsequent statements (top middle of Fig. 6). Conversely, due to the havoc effect, this mutation in Zest removes the `call(foo)` statement (bottom middle of Fig. 6). 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 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.

4 Closely Related Work

While we discussed EI in Section 2.2 to illustrate the consequences of the havoc effect, EI is not the only technique for context-aware mutations. This section provides background on two additional tools for performing finer-grained mutations in generator-based fuzzing: BeDivFuzz [38] and Zeugma [26]. Readers can skip over to Section 5 if they are familiar with these techniques.

4.1 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 [38]. In particular, BeDivFuzz quantifies the behavioral diversity of input by measuring the effective number of diversely covered branches after executing this input [38]. Driven by this diversity feedback, BeDivFuzz adapts its mutation strategies to generate new inputs.

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 Fig. 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 Fig. 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 Fig. 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.

4.2 Smart Crossover in Zeugma

Crossover is an effective technique to generate new input by combining parts of multiple inputs together [1, 7, 13, 20, 24, 44]. 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 Fig. 1) and performs crossover using bytes consumed by the same method.

For each saved input, Zeugma generates a *parametric call tree*, decomposing the bytestream into nodes based on the method call boundaries. Fig. 7 shows the parametric call trees for the inputs in Fig. 2a and Fig. 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 Fig. 7).

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

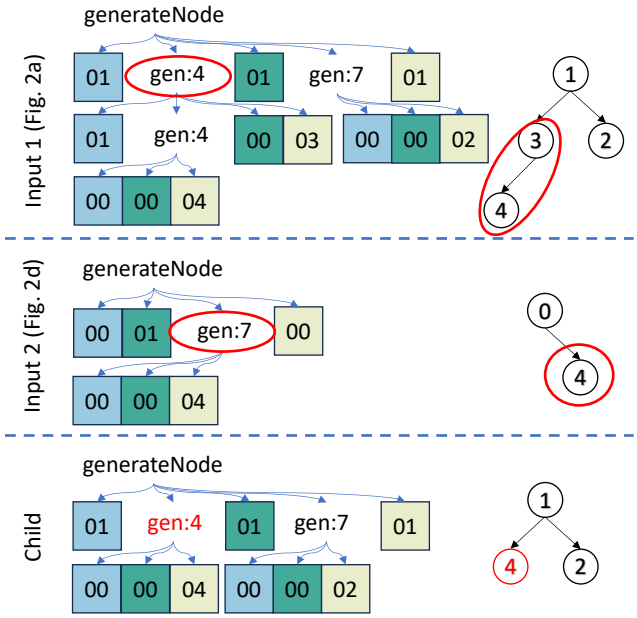


Fig. 7. The parametric call tree for the input 1 (Fig. 2a) and input 2 (Fig. 2d). Note that gen:X represents the method call to generateNode at Line X (Fig. 1c). To perform linked-crossover Zeugma slices based on method call boundaries (e.g. the gen method) and creates a new input by replacing the left child node from input 1 with the right child node from input 2.

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 (c.f. Section 2.1).

5 Evaluation

The goal of our evaluation is to determine how the havoc effect influences the performance of generator-based fuzzers, identify the conditions under which it negatively impacts code coverage, and assess whether existing structure-preserving techniques effectively mitigate this effect. Our evaluation is structured around the following research 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 input validity?
RQ3: How do the mutations performed by our studied techniques impact overall code coverage?
RQ4: Does the additional complexity in our studied techniques affect fuzzing throughput?

In order to answer these questions, we follow the experimental procedures below.

Techniques and Baselines. We evaluate Zest [40], EI, BeDivFuzz [38], and Zeugma [26], as well as two baselines: Random and Zest-Mini. The Random baseline is a modification of Zest that always generates a completely new child byte sequence rather than performing a mutation of the parent input byte sequence, similar to QuickCheck. Zest-Mini is a variant of Zest that only performs 1 mutation with a mean mutation distance of 4 bytes on the parent byte sequence, as opposed to the default Zest settings that apply an average of 8 mutations of 4 bytes each on the parent byte sequence. To isolate the performance overhead introduced by Zeugma's linked-crossover technique,

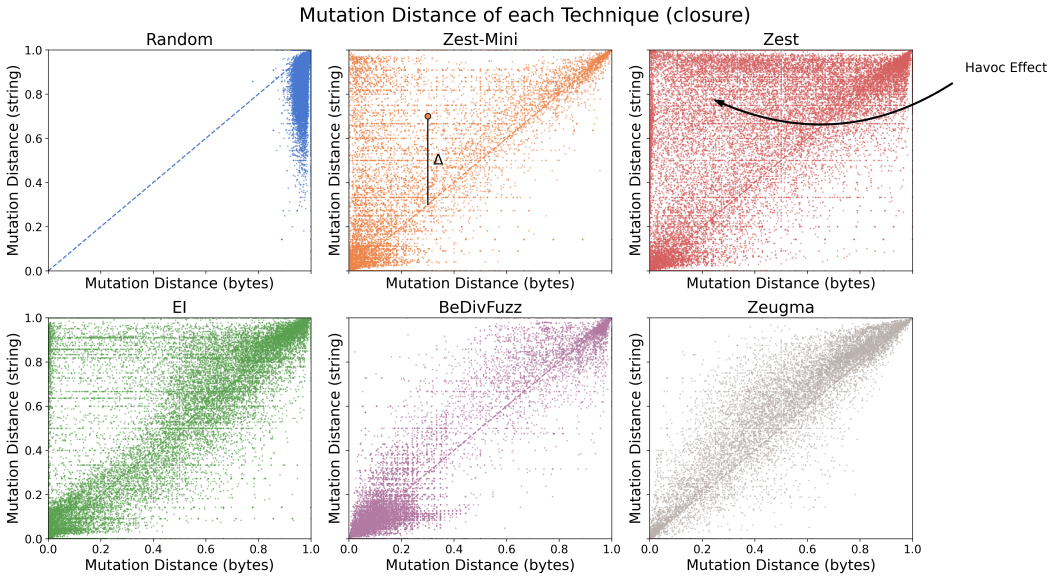


Fig. 8. Scatterplot of mutation distance (between parameter byte sequences on the X-axis, and between the corresponding test inputs generated by the fuzzer on the Y-axis) for the Google Closure benechmark. We observe that techniques that perform context-aware mutations (EI, BeDivFuzz, Zeugma) exhibit less of the havoc effect, i.e. inputs that have a small byte mutation distance and high string mutation distance. Δ measures how correlated the mutation on the parameter byte sequence is with the mutation on the resulting string input.

Table 1. Characteristics of our benchmark applications.

Benchmark	Generator	LOC	Gen-LOC
Ant [2]	XML	140k	136
Maven [37]	XML	93K	136
Rhino Compiler [10]	JavaScript	110K	250
Closure Compiler [4]	JavaScript	250K	250
Chocopy [42]	Python	6K	397
Gson [5]	Json	26K	89
Jackson [6]	Json	49K	89

we also include Zeugma-X [26], which eliminates crossover and splicing. We include Zeugma-X in our evaluation solely to analyze the performance overhead of Zeugma's instrumentation technique and, therefore, do not report its code coverage results.

Benchmarks. We measure the impact of the havoc effect on 7 real-world Java programs with 4 different program generators. Each of the generators and test drivers are reused from prior work [40, 49, 50]. Table 1 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.

Duration. To measure edit distance between inputs, we use a time bound of 1 hour for each experiment. To measure code coverage and performance of each technique, we use a time bound of 24 hours.

Repetitions. Accounting for randomness in fuzzing campaigns, we repeat each experiment 20 times and report aggregated statistics.

Metrics. In our evaluation, we would like to understand the impact of the havoc effect on input edit distance, input validity (as defined by the fuzz driver), and code coverage. For RQ1, we report the *mutation distance* for each technique, defined as the Levenshtein edit distance between the parent input and the child input resulting from the mutation. Since inputs are represented as a bytestreams for the parametric generator and strings for the target application, we report mutation distance for both input representations. For RQ2, we report the frequency at which mutations from each technique result in *valid* inputs, as specified by the test driver. For RQ3 and RQ4, we report branch coverage statistics and runtime overhead[2].

5.1 RQ1: Mutation Distance

First, we would like to understand how destructive the mutations from each of our evaluated techniques are. To evaluate this, we compute the normalized Levenshtein distance between a mutated (child) 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. We run each technique for 1 hour and calculate mutation distances for all inputs (filtering out empty string inputs).

Recall that inputs are represented as a sequence of bytes upon which mutations are performed (ref. Section 2). To measure the impact of the havoc effect on the inputs comprehensively, we need to evaluate both the byte-level mutations and their resulting changes in the generated string inputs. Fig. 8 presents a scatterplot visualization that correlates the normalized mutation distance of parameter byte sequence (X-axis) with the corresponding normalized mutation distance of the string input produced by the generator. For the sake of space, we provide this visualization for just the Closure benchmark; the plots for the rest of the benchmarks can be found Appendix A. This visualization reveals not only the distribution of byte-level mutation sizes across techniques but also how these modifications translate to changes in the final string representation. We can then identify techniques that make relatively small changes to byte sequences yet produce disproportionately large changes in the generated inputs (indicating a strong havoc effect). We include a dashed line of slope 1 as reference to indicate where we expect each of these points to be along; we generally expect mutations to have similar byte-level mutation distance as string mutation distance.

In Fig. 8, we can observe that the techniques performing context-aware mutations (EI, BeDivFuzz, Zeugma) generally feature fewer points in the top left corners of the plots than Zest. This suggests that these techniques, compared to these baselines, are able to do more targeted mutations and produce similar inputs when mutating a small number of bytes. We also note that BeDivFuzz displays a somewhat bimodal distribution, either only mutating a small or large percentage of bytes. Zeugma exhibits a much more uniform distribution of byte-level mutation distance. While EI shows more points closer to the dashed line, a cluster of points along the Y-axis still shows some persistence of the havoc effect. When looking at the plots of the baseline techniques, we note that all the points for the Random baseline live on the right side of the plot by design; a new child input

[2]When reporting mutation distance for the Random baseline, we measure the distance between the parent input and a randomly generated child input. Although the Random baseline does not perform "mutations", we include its mutation distance to provide a baseline that performs completely destructive mutations to sample new byte sequences.

is generated without mutating the parent byte sequence, which would result in an input with high byte-level mutation distance. We interestingly observe that some inputs that have high byte-level mutation distance have much smaller string mutation distance. For Zest-Mini, we observe that limiting Zest to one mutation of 4 bytes does not mitigate the havoc effect substantially, as the distribution of points is fairly similar to that of Zest .

In the plot for Zest-Mini in Fig. 8, we also illustrate the distance from the dashed line with slope $n = 1$. This provides an indication of whether the mutations are *controlled*, i.e. whether the size of the mutation on the parameter bytestream is correlated with the resulting input mutation. So, we define Δ, *normalized mutation distance difference*, as the difference between string-level mutation distance and byte-level mutation distance (string distance minus byte distance). Positive values of Δ indicate the presence of the havoc effect, where mutations to the parametric bytestream cause relatively larger mutations to the string.

Fig. 9 shows the average Δ over all points for each technique-benchmark pair. We expect that more sophisticated techniques exhibit Δ closer to zero. First, we observe that Zest exhibits consistently large positive values, showing that even small byte-level changes frequently result in large string mutations. EI, BeDivFuzz, and Zeugma demonstrate more balanced mutation control, with values closer to zero in many benchmarks. This suggests that these techniques successfully implement context-aware mutations that result in more proportional changes to the final string representation. BeDivFuzz particularly stands out with its consistently low values across benchmarks like Closure and Rhino. Random shows a lower average Δ compared to Zest because its higher byte-level mutation distances lead to predominantly negative Δ values (Fig. 8).

Interestingly, we also observe benchmark-specific variations. For Maven, most techniques seem to avoid the havoc effect (showing negative values). BeDivFuzz exhibits high Δ for Gson and Jackson compared to the other benchmarks. This highlights how the havoc effect's magnitude depends not only on the mutation technique but also on the input generator and the target application.

➲ **Finding #1**: Techniques performing context-aware mutations (EI, BeDivFuzz, Zeugma) exhibit more control over mutations than the coverage-guided fuzzing baselines (Zest, Zest-Mini). This results in a decrease in the havoc effect.

In our measurements, we also noticed a number of "zero mutations" for each technique—-cases where a mutation produces a child input identical to its parent. Zero mutations occur when the mutation to the input bytestream does not change the generator-produced input, effectively creating a duplicate input. For example, in Fig. 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).

Fig. 10 shows the percentage of zero mutations for all techniques on each benchmark. We first note that the random baseline achieves close to no zero mutations across all benchmarks. Zest-Mini, which performs a single mutation of 4 bytes, displays a significantly higher percentage of zero mutations; this is expected, as the byte-level representation of the child inputs are much more similar to their parents compared to the other techniques. So, while Zest-Mini does somewhat mitigate the havoc effect (ref. Fig. 9), it appears to do this with the price of more zero mutations than the other techniques.

We also notice a significantly higher percentage of zero mutations in all techniques when testing ChocoPy. This suggests that the ChocoPy generator is implemented in a way such that a significant percentage of parametric inputs are decoded to the exact same string input. We recommend that researchers and practitioners should track the zero mutation rate as a method of debugging and improving generators.

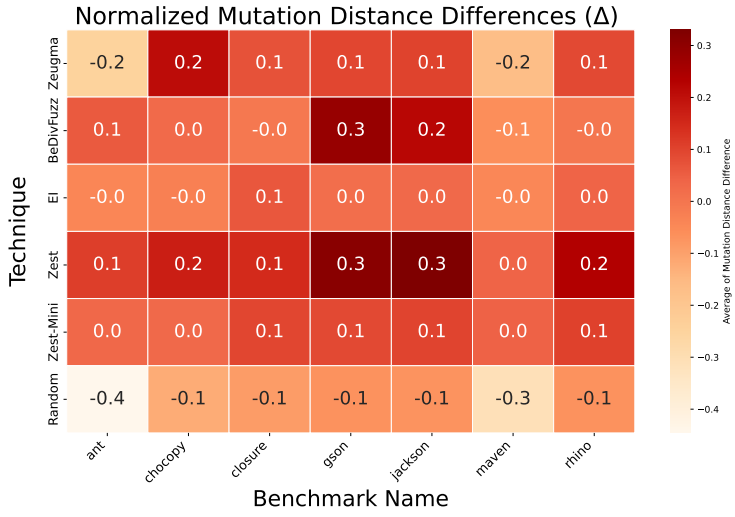


Fig. 9. Heatmap of mutation distance difference, with the benchmark on the X-axis and technique on the Y-axis (values closer to zero are better). We observe that techniques that perform context-aware mutations (EI, BeDivFuzz, Zeugma) demonstrate better control over mutations.

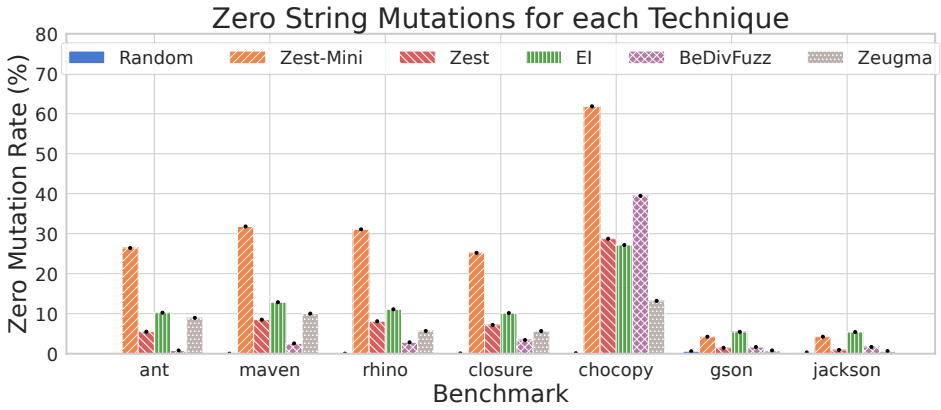


Fig. 10. Percentage of zero mutations (i.e., children identical to parent) out of all generated inputs for the studied techniques. Techniques that mutate a smaller number of bytes tend to produce a higher percentage of zero mutations. Lower is better.

Finding #2: Techniques mutating a relatively smaller number of bytes (Zest-Mini) produce a higher percentage of zero mutations. Context-aware mutation techniques exhibit a similar zero mutation rate as Zest. The implementation of the generator also affects the frequency of zero mutations across all techniques.

5.2 RQ2: Validity-Preserving Mutations

Next, we want to understand which techniques are able to preserve *input validity* when performing mutations. Validity of inputs for each benchmark is defined in its corresponding fuzz driver; these

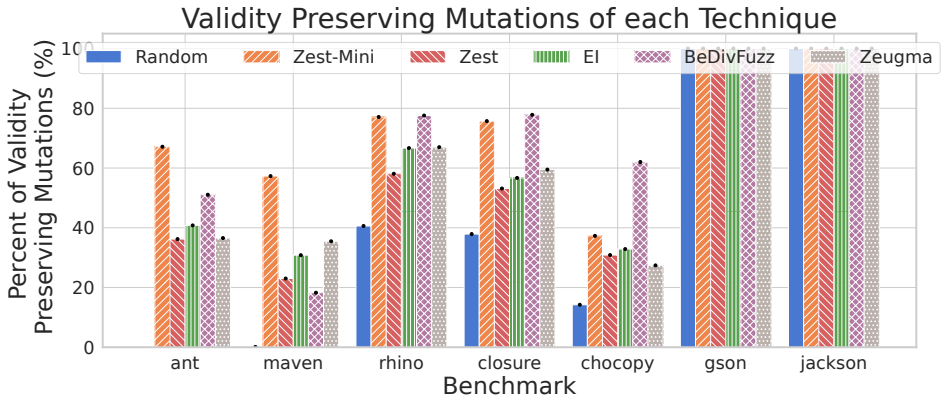


Fig. 11. Percentage of validity-preserving mutations for all studied techniques. Higher is better.

include syntactic checks for JSON and semantic checks for ChocoPy, Javascript, and XML. For each technique, we measure the percentage of mutations applied on valid parent inputs that result in valid child inputs.

Fig. 11 shows the percent of validity-preserving mutations (filtering out zero mutations) for each technique, across all benchmarks. The bars for the Random baseline provide a measurement of how often the input generator, on its own, is able to generate valid inputs. For Gson and Jackson, since the validity check is only syntactic, the random JSON generator always produces valid JSON inputs. Thus, the validity preserving mutation rate is 100% for all techniques.

In general, all of the mutation-based techniques achieve a higher percentage of validity-preserving mutations than the random baseline. The techniques producing inputs with lower mutation distance (Zest-Mini, BeDivFuzz) achieve a higher percent of validity-preserving mutations than the other techniques. This is expected, as child inputs are more similar to parent inputs and are more likely to retain features relevant to validity. Similarly, EI generates inputs with higher percentage of validity than Zest. As shown in Fig. 9, EI demonstrates better control over mutation distances, thus having less destructive mutations than Zest. Therefore, it is not surprising that the mutations in EI better preserve input validity.

Finding #3: Parametric mutation-based fuzzing techniques substantially improve input validity compared to random generation. Our analysis shows that while the studied techniques perform similarly overall, those that modify fewer bytes in the parametric input tend to better preserve validity across mutations.

5.3 RQ3: Coverage Comparison

RQ3 examines how different fuzzing techniques compare in terms of the *coverage* achieved across each benchmark. We report the coverage achieved by all techniques over 24-hour experiments. Since the techniques are implemented in different frameworks with their own instrumentation, we include both a *trial-bounded* and *time-bounded* comparison.

The time-bounded comparison runs each fuzzing campaign for 24 hours. For the trial-bounded comparison, we take the minimum number of inputs generated across all campaigns and truncate all coverage results to that number.

Since Zeugma is implemented in a framework with faster instrumentation compared to the JQF-based tools, we also introduce the baseline Zeugma^N, which normalizes Zeugma's coverage using a three-step process: (1) recording Zest's total execution count, (2) measuring the time Zeugma-X needs to reach this count, and then (3) determining how much coverage Zeugma reaches within that same timeframe. Our comparative analysis serves two complementary purposes: the trial-bounded branch coverage approximates each technique's potential performance without implementation overhead, while Zeugma^N estimates coverage if Zeugma also uses Zest's instrumentation framework. Our final evaluation employs branch coverage collected via JaCoCo (version 0.8.7), following recommendations from prior literature [26, 35].

Fig. 12 displays median branch coverage over time for all techniques. The shaded areas represent the min and max code coverage across 20 repetitions. Table 2 shows the median branch coverage for both trial-bounded and time-bounded comparisons. We first observe that the Random baseline performs significantly worse on all benchmarks than Zest. This is expected, as the input validity constraints are fairly strict. Zest-Mini shows no statistical difference in branch coverage compared to Zest, in all benchmarks but ChocoPy.

We find fairly mixed results across the techniques computing context-aware mutations. EI is able to achieve the same or significantly better results than Zest all benchmarks. From RQ1, we note that EI generally exhibits less of the havoc effect than Zest. This suggests that more targeted mutations can be beneficial for exploring the input space. Looking at the time-bounded comparison, EI achieves significantly higher branch coverage in ChocoPy and Closure, even with the execution overhead of performing these context-aware mutations.

Interestingly, BeDivFuzz achieves significantly less branch coverage than Zest in all benchmarks but Gson and Jackson. While this is partially due to the execution overhead in BeDivFuzz, we see that the branch coverage is significantly lower even in the trial-bounded comparison. This suggests that more destructive mutations are still desirable for exploring new code paths. We believe BeDivFuzz is essentially filtering down the Zest mutations to the least-destructive ones, whereas EI and Zeugma are introducing finer-grained mutations while retaining still more destructive mutations.

When looking at the branch coverage achieved by Zeugma, we note that the tool is implemented in a different framework that performs faster instrumentation than the JQF-based tools. Thus, the fixed-trial branch coverage and Zeugma^N provide a fairer comparison to the other techniques. We find that Zeugma is able to achieve significantly greater median coverage than Zest in four out of seven benchmarks. On the Javascript-based targets, Zeugma performs significantly better than EI on Rhino. On Closure, however, while Zeugma shows better performance than EI, its normalized version (Zeugma^N) performs worse when accounting for execution speed differences. Zeugma achieves a significantly lower branch coverage than EI on Ant but significantly higher on Maven. Looking at Fig. 9 from Section 5.1, Zeugma has an average mutation distance difference of -0.2 for both of these XML-based targets. This suggests that it depends on the target program on whether the destructiveness of the mutation helps with code coverage. The trial-bounded coverage measurement and Zeugma^N also highlight that the branch coverage is influenced by confounding variables such as the execution speed of the underlying fuzzing framework, which can mask a technique's true effectiveness when code coverage is compared directly. We discuss these findings further in Section 8.

➲ **Finding #4**: Techniques performing context-aware mutations (EI, Zeugma) can result in significantly higher code coverage than coverage-guided fuzzing (Zest). However, incorporating

Table 2. For each fuzzer, we report the median branch coverage in application classes for each subject across 20 fuzzing campaigns after 24 hours. Values in red indicate a statistically significant decrease compared to Zest, while olive values show a significant increase. The highest value for each benchmark is highlighted in blue.

Fuzzer	Ant		Chocopy		Closure		Gson		Jackson		Maven		Rhino	
	Fixed Trial	24hr	Fixed Trial	24hr	Fixed Trial	24hr	Fixed Trial	24hr	Fixed Trial	24hr	Fixed Trial	24hr	Fixed Trial	24hr
Random	622.5	755.0	876.0	876.0	10908.5	11227.0	319.0	319.0	1112.0	1113.0	600.0	669.5	3162.5	3227.0
Zest-Mini	911.5	914.0	875.0	876.0	12407.5	12443.5	321.0	321.0	1121.0	1121.0	1085.5	1138.0	3514.5	3631.5
Zest	919.5	920.5	889.0	889.0	12405.0	12520.0	321.0	321.0	1121.0	1121.0	1100.5	1138.0	3474.5	3542.5
EI	918.0	920.5	898.5	899.0	12679.5	12853.5	321.0	321.0	1121.0	1121.0	1117.5	1138.0	3459.5	3494.0
BeDivFuzz	888.5	891.5	880.0	889.0	12209.5	12373.0	321.0	321.0	1121.0	1121.0	683.5	697.0	3286.0	3310.0
Zeugma	904.0	908.5	877.0	877.0	12707.5	13406.5	321.0	321.0	1123.0	1123.5	1138.0	1138.0	3798.5	3852.0
Zeugma^N	N/A	903.5	N/A	877.0	N/A	12737.5	N/A	321.0	N/A	1123.5	N/A	1138.0	N/A	3838.0

more destructive mutations is also important for exploring new code paths, as BeDivFuzz consistently achieves lower coverage than Zest.

We also would like more closely understand how the mutations performed by our evaluated techniques relate to code coverage. In particular, we would like to understand how mutation distance relates to coverage increases. We compare the mutation distances of all generated inputs ("All") the subset of inputs that increased code coverage ("Saved") during our one hour experiments for all evaluated techniques. Fig. 13 shows the ratio of saved input median mutation distance to all input mutation distance for all technique-benchmark pairs. We observe that the ratio is above 1 for EI and BeDivFuzz across all benchmarks, suggesting that more destructive mutations are more beneficial for producing coverage-increasing inputs. Interestingly, Zeugma shows the opposite trend, where mutations that produce saved inputs are significantly less destructive than the median mutation over all inputs. This can be potentially explained by Zeugma's approach of introducing random mutations on top of their crossover splicing approach with 50% probability (ref. Section 4). Zest shows a ratio of close to 1, suggesting that the average mutation distance is not significantly different than the mutation distances resulting in saved inputs. We believe this type of analysis can help tune various parameters in the fuzzing algorithm, such as the probability that more destructive mutations are performed.

➲ **Finding #5**: For BeDivFuzz and EI, the median mutation distance of mutations that produce coverage-increasing inputs exceeds the median mutation distance of all inputs, while for Zeugma, the opposite holds. This suggests that BeDivFuzz and EI may benefit from more destructive mutations, whereas Zeugma may benefit from fewer.

Notably, Zeumga is the only technique that combines both random byte mutation with context-aware mutations, which potentially explains its higher coverage across most benchmarks. We believe this is the smartest approach to use in practice: the random byte mutations enable the fuzzer to broadly explore diverse regions of the target program, while the context-aware mutations facilitate targeted exploitation to cover deeper code paths. Moreover, as described in Section 3, context-aware mutations demonstrate particular effectiveness in covering program locations that process structured input in non-sequential order. Future work could investigate a strategic integration of these complementary tactics for exploration and exploitation.

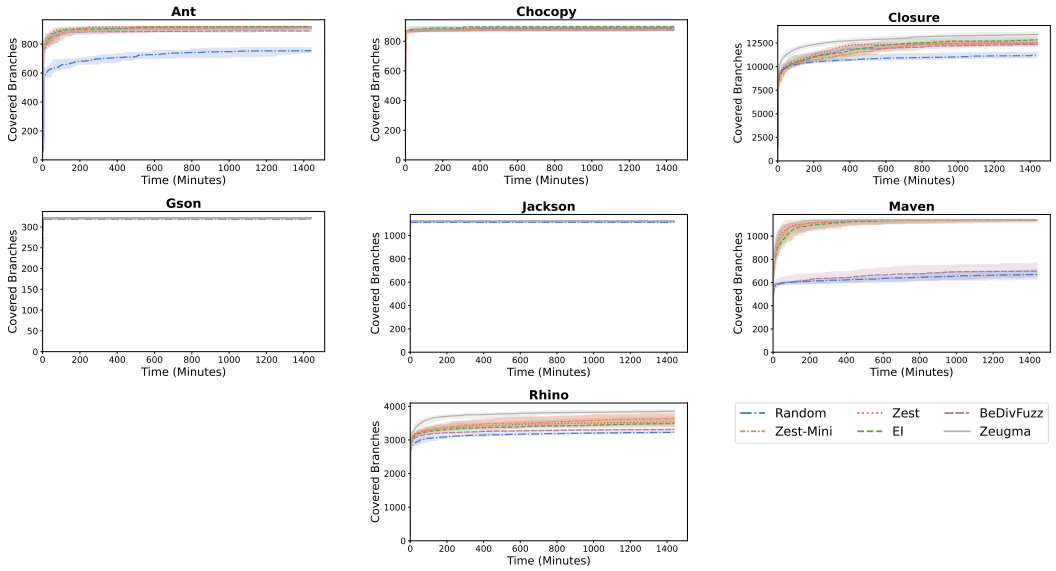


Fig. 12. Median branch coverage over time (X axis is time, Y axis is number of branches covered) over 24 hours, with the shaded region representing the entire range across 20 repetitions.

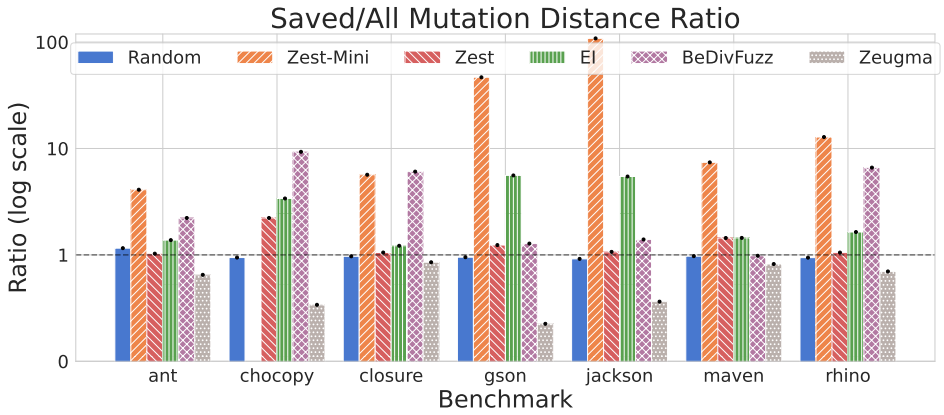


Fig. 13. Ratio of median mutation distance of saved inputs (those that increase code coverage) to median mutation distance of all generated inputs. Values higher than 1 suggest that more destructive mutations result in inputs that increase code coverage.

5.4 RQ4: Runtime Overhead

While we observe that techniques performing context-aware mutations are generally able to achieve higher code coverage than the baselines, they often come at the cost of overhead. We would like to understand this runtime overhead for each of our techniques. Table 3 shows the geometric mean execution slowdown of inputs-per-second (24 hours, 20 repetitions) of the context-aware fuzzing techniques compared to coverage-guided fuzzing. Since EI and BeDivFuzz are JQF-based tools, we compare their execution speed to Zest. We compare Zeugma to a variant called Zeugma-X [26],

Table 3. Geometric means of runtime slowdown of each technique, compared to coverage-guided fuzzing baseline. Values closer to 0 indicate more runtime overhead of the technique.

Benchmark	EI	BeDivFuzz	Zeugma
ant	$0.83 \times 1.18^{\pm 1}$	$0.93 \times 1.19^{\pm 1}$	$0.56 \times 1.16^{\pm 1}$
chocopy	$1.01 \times 1.18^{\pm 1}$	$2.11 \times 1.55^{\pm 1}$	$0.67 \times 1.11^{\pm 1}$
closure	$0.85 \times 1.10^{\pm 1}$	$0.76 \times 1.10^{\pm 1}$	$0.66 \times 1.13^{\pm 1}$
gson	$0.54 \times 1.16^{\pm 1}$	$0.55 \times 1.32^{\pm 1}$	$0.85 \times 1.08^{\pm 1}$
jackson	$0.74 \times 1.25^{\pm 1}$	$0.77 \times 1.17^{\pm 1}$	$0.96 \times 1.14^{\pm 1}$
maven	$0.43 \times 1.15^{\pm 1}$	$0.20 \times 1.27^{\pm 1}$	$0.72 \times 1.08^{\pm 1}$
rhino	$0.64 \times 1.12^{\pm 1}$	$0.83 \times 1.17^{\pm 1}$	$0.90 \times 1.04^{\pm 1}$

which eliminates crossover and splicing. Values closer to zero (red) indicate more runtime overhead, and values above one (green) indicate a faster execution speed than the baseline.

Our results reveal some interesting patterns regarding the runtime overhead of different fuzzing techniques. Generally, the context-aware mutations in EI and BeDivFuzz result in significant slowdown, with the notable exception of BeDivFuzz in the ChocoPy benchmark. BeDivFuzz demonstrates anomalously high execution speed in ChocoPy. This is likely due to the fact that BeDivFuzz only saves valid inputs and thus may perform (faster) random exploration for a significant portion of the fuzzing campaign. We note that although BeDivFuzz achieves speedup on ChocoPy over Zest, this may be due to executing less code in ChocoPy, as shown by its lower coverage in RQ3.

Overall, while the sophisticated techniques generally exhibit some runtime overhead compared to the coverage-guided baseline, the magnitude varies significantly by benchmark. This suggests that the efficiency of these techniques is highly dependent on the characteristics of the target application. When considering these techniques or practical applications, users should weigh the potential coverage gains against the performance impact for their specific context. We also strongly encourage researchers to publish execution slowdown of each of the evaluated techniques to provide more information about this tradeoff.

➲ **Finding #6**: Techniques performing context-aware mutations incur significant runtime overhead compared to regular parametric coverage-guided fuzzing. The choice of saving criteria can also affect the overall execution speed throughout the campaign.

6 Threats to Validity

6.1 Construct Validity

In our study, we employ code coverage to quantify the effectiveness of different mutation strategies. Leading fuzzing researchers continue to question whether bug detection rate or code coverage are better predictors of finding new bugs [12, 15, 18] with coverage being slightly more consistent as a predictor [12]. In the context of understanding whether havoc mutations prevent the fuzzer from exploring certain branches in real-world applications, as illustrated in Fig. 5, we believe that code coverage is an adequate measurement. Additionally, when the goal is to synthesize a fuzz corpus for regression testing, coverage is more important and a necessary condition for finding yet-unintroduced bugs.

Another threat to construct validity is our use of Levenshtein distance to measure mutation distance. While this edit distance metric provides a generic approach to automatically quantify mutations in string-based representations, it may not accurately capture semantic changes in structured data, potentially leading to imprecise measurements. For example, in tree-structured

inputs such as code programs, a tree-based mutation distance metric might better capture the relationships between mutations and their effects on target behavior. The Levenshtein distance treats all character-level edits equally, potentially missing the hierarchical importance of changes in structured data formats.

6.2 Internal Validity

A threat to internal validity in our study is our method of measuring fuzzer overhead using trials per second. This metric is influenced not only by the computational overhead of the fuzzer itself but also by whether the generated inputs cause the program to execute slower code paths. The variations in trials per second might therefore be attributed to differences in execution paths rather than differences in fuzzer efficiency. This makes it challenging to isolate the true computational overhead introduced by different mutation strategies.

6.3 External Validity

Our implementation is based on JQF [39], and all target applications and their input generators are written in Java. Our conclusion may not be generalized to generator-based greybox fuzzers targeting other programming languages such as libFuzzer [7] and cargo-fuzz [3]. Additionally, our research focuses on target applications that process inputs represented as strings. This choice was made to facilitate the comparison of editing distances for generator-produced inputs. However, this focus on string inputs might not extend to applications dealing with binary inputs, such as PDF or image viewers.

7 Additional Related Work

Havoc mutations. Researchers and practitioners have introduced a havoc phase into grey-box fuzzers to assist in escaping local minima [20, 33, 53, 55]. This approach has proven effective in several studies and has been widely adopted by many state-of-the-art fuzzers [30, 35]. Kukucka et al. [31] study perform an empirical study on the impact of various mutators in AFL++, including the havoc mutation. It is important to distinguish the *havoc mutation* from the *havoc effect*, as the latter refers to unintended behavior unique to generator-based fuzzing.

Grammar-based input generation. There is a rich body of work focusing on grammar-based input generation for greybox fuzzing [9, 13, 16, 23, 47, 48, 56]. 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 [48]. However, the effectiveness of such a mutation strategy has been questioned in subsequent research [21], 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 the 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 [43], Tzer [34], Bonsai fuzzing [50], and FuzzChick [32]; and (2) employing dynamic analysis to pinpoint bytes of interest for mutation, as seen in approaches such as Confetti [29], GreyOne [22], and Angora [17]. Specifically, DIE [43] enhances mutation effectiveness by preserving "interesting" types and structures, which focuses the search process. In a similar vein, Tzer [34] integrates tensor-compiler-specific mutations—tailored for deep learning systems—with general-purpose mutation strategies to achieve a balanced mix of

exploration and exploitation. Bonsai fuzzing [50], initially designed for generating smaller inputs, also enhances fuzzing precision by gradually increasing the hyper-parameters that govern input size. 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.

8 Reflections

In this paper, we conducted a thorough evaluation of multiple fuzzing techniques across many benchmarks. This required significant effort to look into the internal implementations of different fuzzing techniques to log various intermediate results such as mutation distance, zero mutations, and other metrics. Throughout this experience, we gained valuable insights that we hope can benefit the fuzzing research community.

A fundamental question remains challenging to answer definitively: how can we determine if a new fuzzing algorithm is "better" than existing approaches? This question is complex due to the many confounding factors in fuzzing experiments that make it easy to draw incorrect conclusions. Without careful consideration of these factors, researchers risk making claims that do not generalize or may even be misleading.

First, how do we know that a new fuzzing algorithm is implemented correctly? The feedback loop for fuzzing algorithms has a high cost, meaning it often takes considerable time—sometimes days—to get results for a new implementation. There can also be subtle bugs in implementation that may not surface in end-to-end metrics like code coverage. Although it may seem obvious, we argue that fuzzing researchers should perform proper sanity checks for their implementations. From our experience, examining intermediate-level metrics like mutation distance was able to surface issues that would otherwise remain hidden in overall performance measurements, such as the presence of zero mutations.

Second, how do we evaluate whether a new fuzzing algorithm outperforms other approaches? Prior work [28, 46, 52] has advanced evaluation practices to better make statistical claims about algorithms, ensuring multiple repetitions, appropriate statistical tests, seed control, and more. However, many other factors remain to consider. As seen from our results, the best-performing fuzzing algorithm can change across target applications. For researchers and practitioners looking to use and evaluate fuzzing algorithms, a deeper analysis of intermediate results is valuable in understanding the effectiveness of each technique. We urge researchers to perform and publish this analysis so that the community can better isolate the reasons certain algorithms perform better on different benchmarks.

We concretely recommend the following two practices:

(1) Reporting execution overhead of algorithms across all benchmarks. We saw that algorithms had relatively different overheads, depending on the benchmark.
(2) Reporting count-based coverage measurements along with time-based coverage. This can help isolate whether improvements in coverage are due to faster implementations or smarter search strategies in the input space.

9 Conclusion

In this paper, we investigated the challenges associated with generator-based coverage-guided greybox fuzzing, particularly focusing on the havoc effect, where small bytestream mutations result in disproportionate changes in generator-produced inputs. We perform a thorough evaluation across seven real-world Java programs measuring the impact of the havoc effect on various generator-based fuzzing techniques, including context-aware mutation strategies such as JQF's EI, BeDivFuzz,

and Zeugma. By introducing the notion of mutation distance, measured as the Levenshtein distance between the parent input and mutated child input, we are able to quantify the havoc effect in each of the studied techniques. Our evaluation results find that context-aware mutation strategies are able to significantly reduce the havoc effect and better preserve input validity than Zest, our coverage-guided fuzzing baseline. But is the havoc effect actually good or bad? Our empirical results indicate that the havoc effect is likely beneficial for initial exploration, but limiting this effect allows exploitation to cover some deeper program paths. We also find that context-aware mutation strategies incur significant runtime overhead, highlighting a practical tradeoff between performance and speed. Techniques such as Zeugma that combine destructive and context-aware mutations seem to perform very well overall. Our work provides insights into the nuanced interplay between input mutation strategies and application characteristics that we hope will benefit future researchers and practitioners in this field.

Acknowledgement

We would like to thank the anonymous reviewers from the FUZZING'24 workshop for their valuable feedback on this paper. We thank Jonathan Bell, Michael Hicks, and Addison Crump for their insightful discussion that helped improve this work.

This material is based upon work supported by the National Science Foundation (NSF) CCF-2120955, Defense Advanced Research Projects Agency (DARPA), and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract NN66001-22-C-4027. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF, DARPA, or NIWC Pacific.

References

[1] 2024. American Fuzzy Lop. https://github.com/google/AFL. Accessed: 2024-12-15.
[2] 2024. Apache Ant is a Java-based build tool. https://github.com/apache/ant.
[3] 2024. cargo-fuzz. https://github.com/rust-fuzz/cargo-fuzz. Accessed: 2024-12-15.
[4] 2024. Closure Compiler. https://developers.google.com/closure/compiler. Accessed: 2024-12-15.
[5] 2024. GSON: A Java serialization/deserialization library to convert Java Objects into JSON and back. https://github.com/google/gson.
[6] 2024. Jackson Project Home @github. https://github.com/FasterXML/jackson.
[7] 2024. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. Accessed: 2024-12-15.
[8] 2024. libFuzzer – How To Split A Fuzzer-Generated Input Into Several Parts. https://github.com/google/fuzzing/blob/41d7725/docs/split-inputs.md. Accessed: 2024-12-15.
[9] 2024. libprotobuf-mutator. https://github.com/google/libprotobuf-mutator. Accessed: 2024-12-15.
[10] 2024. Rhino: JavaScript in Java. https://github.com/mozilla/rhino.
[11] 2024. Structure-Aware Fuzzing with libFuzzer. https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md. Accessed: 2024-12-15.
[12] 2024. Thinking about Fuzzer Evaluation. https://addisoncrump.info/research/thinking-about-fuzzer-evaluation/. Accessed: 2024-12-15.
[13] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars.. In *NDSS*.
[14] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering* 45, 5 (2017), 489–506.
[15] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the reliability of coverage-based fuzzer benchmarking. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1621–1633. https://doi.org/10.1145/3510003.3510230
[16] 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

[17] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.
[18] Yiqun T. Chen, Rahul Gopinath, Anita Tadakamalla, Michael D. Ernst, Reid Holmes, Gordon Fraser, Paul Ammann, and René Just. 2020. Revisiting the Relationship Between Fault Detection, Test Adequacy Criteria, and Test Set Size. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 237–249.
[19] 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
[20] 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.
[21] 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
[22] 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.
[23] 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
[24] 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
[25] Paul Holser. [n. d.]. junit-quickcheck: Property-based testing, JUnit-style. https://github.com/pholser/junit-quickcheck. Accessed: 2024-12-15.
[26] Katherine Hough and Jonathan Bell. 2024. Crossover in Parametric Fuzzing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.
[27] 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
[28] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2123–2138.
[29] 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.
[30] James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. 2024. An Empirical Examination of Fuzzer Mutator Performance. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1631–1642. https://doi.org/10.1145/3650212.3680387
[31] James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. 2024. An Empirical Examination of Fuzzer Mutator Performance. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1631–1642.
[32] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage guided, property based testing. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 181 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360607
[33] 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.
[34] 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
[35] 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
[36] Charlie Miller, Zachary NJ Peterson, et al. 2007. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep* 4 (2007).
[37] Frederic P Miller, Agnes F Vandome, and John McBrewster. 2010. *Apache Maven*. Alpha Press.
[38] 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
[39] 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
[40] 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.
[41] 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
[42] 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
[43] 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
[44] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering* (2019).
[45] 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.
[46] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. Sok: Prudent evaluation practices for fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1974–1993.
[47] 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
[48] 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
[49] 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
[50] Vasudev Vikram, Rohan Padhye, and Koushik Sen. 2021. Growing A Test Corpus with Bonsai Fuzzing. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 723–735. https://doi.org/10.1109/ICSE43902.2021.00072
[51] 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.
[52] Dylan Wolff, Marcel Böhme, and Abhik Roychoudhury. 2022. Explainable fuzzer evaluation. *arXiv preprint arXiv:2212.09519* (2022).
[53] 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
[54] 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
[55] 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 18)*. 745–761.
[56] 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

10 Appendix

This appendix contains additional figures.

10.1 Scatterplots for all benchmarks

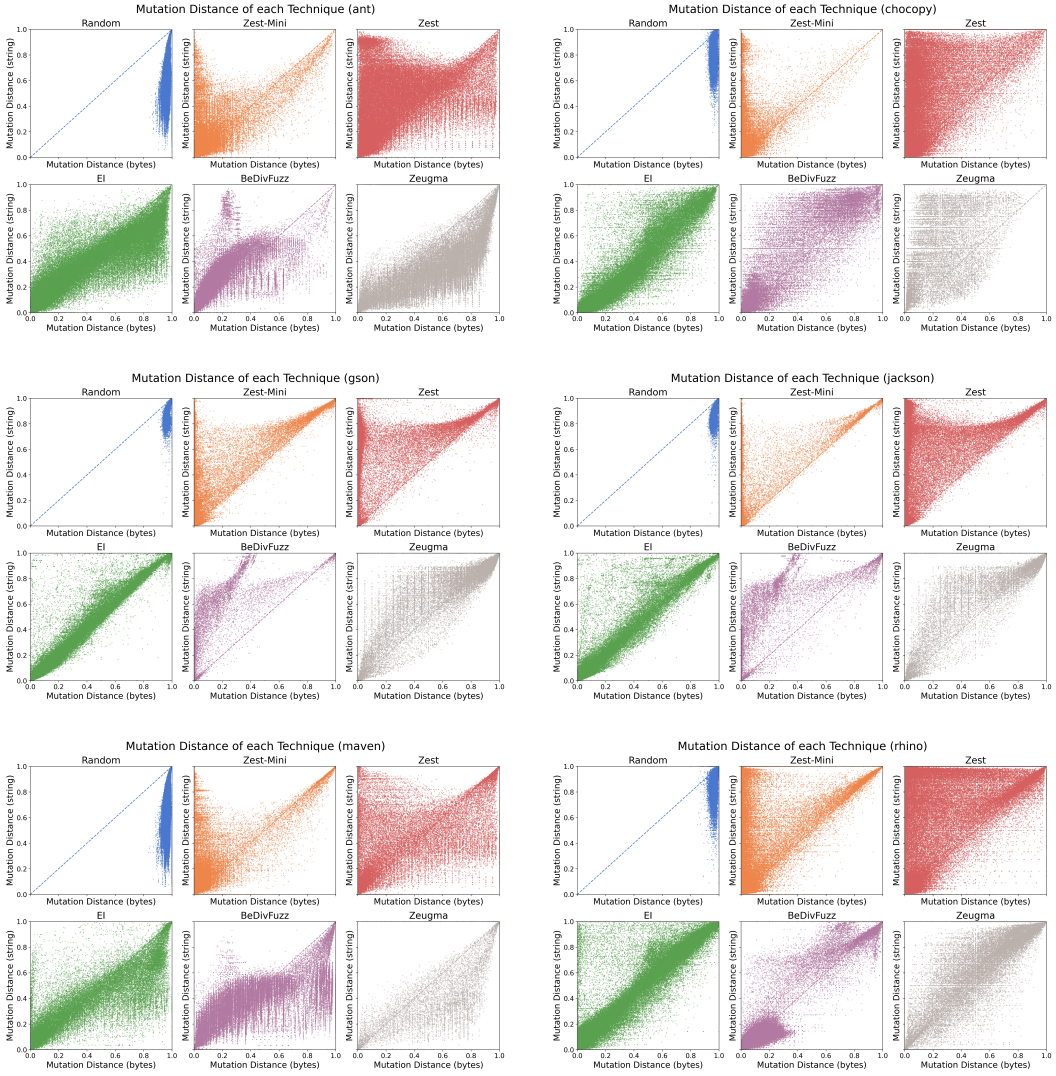


Fig. 14. Scatterplots of mutation distance (bytes on the X-axis, string on the Y-axis) across six target programs. We observe that techniques that perform finer-grained mutations (EI, BeDivFuzz, Zeugma) exhibit less of the havoc effect, i.e. inputs that have a small byte mutation distance and high string mutation distance.