Crossover in Parametric Fuzzing

Katherine Hough
Northeastern University
Boston, Massachusetts, United States
hough.k@northeastern.edu

Jonathan Bell
Northeastern University
Boston, Massachusetts, United States
j.bell@northeastern.edu

ABSTRACT

Parametric fuzzing combines evolutionary and generator-based fuzzing to create structured test inputs that exercise unique execution behaviors. Parametric fuzzers internally represent inputs as bit strings referred to as "parameter sequences". Interesting parameter sequences are saved by the fuzzer and perturbed to create new inputs without the need for type-specific operators. However, existing work on parametric fuzzing only uses mutation operators, which modify a single input; it does not incorporate crossover, an evolutionary operator that blends multiple inputs together. Crossover operators aim to combine advantageous traits from multiple inputs. However, the nature of parametric fuzzing limits the effectiveness of traditional crossover operators. In this paper, we propose linked crossover, an approach for using dynamic execution information to identify and exchange analogous portions of parameter sequences. We created an implementation of linked crossover for Java and evaluated linked crossover's ability to preserve advantageous traits. We also evaluated linked crossover's impact on fuzzer performance on seven real-world Java projects and found that linked crossover consistently performed as well as or better than three state-of-the-art parametric fuzzers and two other forms of crossover on both long and short fuzzing campaigns.

CCS CONCEPTS

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

KEYWORDS

fuzz testing, test input generation, generator-based fuzzing, parametric fuzzing, dynamic analysis

ACM Reference Format:
Katherine Hough and Jonathan Bell. 2024. Crossover in Parametric Fuzzing. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639160

1 INTRODUCTION

Early identification of software defects is crucial for mitigating their impact and reducing the cost of repairing them. Evolutionary fuzzing is a prominent technique for automatically generating test inputs that leverages information about system executions to bias the exploration of an input space towards promising areas in order to maximize the diversity of explored execution behaviors. Over the course of a fuzzing campaign, inputs are "evolved" by maintaining a population of interesting inputs that have been discovered and creating new inputs by perturbing these interesting inputs. This evolutionary search process depends on the assumption that an input created by making a small change to an interesting input is more likely to reveal new execution behaviors than a purely random input. Evolutionary fuzzing has been shown to be effective at finding defects in real world systems [18, 25, 35, 43, 46, 47].

However, if the system under test has a highly constrained input structure, then even small changes to an input are likely to violate those constraints. Thus, an evolutionary fuzzer that is unaware of the system's input structure may struggle to create valid inputs that exercise the core functionality of the system. Padhye et al. [46] proposed and demonstrated the effectiveness of parametric fuzzing, a technique for performing structure-aware, evolutionary fuzzing. Parametric fuzzers use QuickCheck [10]-style generators to achieve structural awareness. In parametric fuzzing, generators are used to map fuzzer-created bit strings, referred to as "parameter sequences", to structures [23, 38, 46, 49]. The parametric fuzzer provides a means of splitting the parameter sequence into arbitrary, primitive-typed values. These arbitrary values are then used by one or more generators to build structured test inputs that conform to user-defined constraints.

The strength of parametric fuzzing is that it supports generators of arbitrary types without requiring developers to define type-specific mutation operators (operators that modify a single input) and crossover operators (operators that combine parts from multiple inputs together) to perturb inputs. Since a parameter sequence is a bit string, parametric fuzzers are able to modify parameter sequences using generic mutation and crossover operators. However, existing parametric fuzzers do not support crossover and use only mutation operators to modify parameter sequences [23, 38, 46, 49].

Prior work has demonstrated the effectiveness of crossover in structured fuzzing [8, 18, 47]. Furthermore, many unstructured fuzzers, such as AFL [35], AFL++ [16], and libFuzzer [25], utilize some form of general-purpose crossover operator. Some fuzzers, like AFL++ [16] and libFuzzer [25], even support custom, user-defined crossover operators. The efficacy of crossover operators stems from their ability to produce children that inherit advantageous traits from multiple parent inputs — a property referred to as heritability [48, 51].

Unfortunately, parametric fuzzing lacks an explicit tree structure rendering tree-based crossovers operators like the ones proposed by Pham et al. [47], Holler et al. [18], and Aschermann et al. [8] inapplicable. Furthermore, the nature of parametric fuzzing limits the effectiveness of unstructured crossover operators like the ones found in AFL and libFuzzer. Our key insight is that a tree structure

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639160

can be extracted from the hierarchy of method calls made by generators in parametric fuzzing. Using this insight, we designed a new crossover operator for parametric fuzzing — "linked crossover". Empirically comparing our approach to traditional crossover operators, we found that linked crossover produces new inputs that inherit more desirable traits from their parents. In a comparison against three state-of-the-art parametric fuzzers, we found that applying linked crossover increased branch coverage and defect detection rates. Overall, this work makes the following contributions:

- A description of linked crossover, a novel crossover operator for parametric fuzzing that leverages call tree information to intelligently combine inputs.
- A open source implementation of linked crossover for Java.
- An empirical comparison of the heritability and effectiveness of different crossover operators in parametric fuzzing.
- An evaluation of the effectiveness of linked crossover on seven Java projects against three state-of-the-art parametric fuzzers: Zest [46], BeDivFuzz [38], and RLCheck [49].

2 BACKGROUND

2.1 Evolutionary Fuzzing

Algorithm 1 depicts a generic, evolutionary fuzzing algorithm that is similar to the ones used by AFL [35] and libFuzzer [25] The evolutionary fuzzer maintains a population of interesting inputs. The fuzzer repeatedly creates and executes new inputs. These inputs are created by selecting a parent input from the population and perturbing that input by applying a number of mutation and crossover operators to produce a child. This child is then executed as an input to the fuzzing target and, the fuzzer observes execution feedback. The type of execution feedback depends on the fuzzer; branch coverage is a common choice [25, 35, 46]. If the child exercises new coverage, it is saved to a corpus of coverage-revealing inputs. If the child induced a new failure, it is saved to a directory of failure-inducing inputs. Lastly, the population may be updated, typically to include this new child input if the child revealed new system behavior.

Algorithm 1 A generic, evolutionary fuzzer.

```
1:  failures ← {}
2:  totalCoverage ← {}
3:  population ← {}
4:  while there is time remaining in the campaign do
5:      if population is empty then
6:          child ← a new random input
7:      else
8:          parent ← select(population)
9:          child ← modify(parent, population)
10:     end if
11:     coverage, feedback, failure ← execute(child)
12:     if ∃x ∈ coverage : x ∉ totalCoverage then
13:         save child to the corpus
14:         totalCoverage ← totalCoverage ∪ coverage
15:     end if
16:     if failure ≠ □ ∧ failure ∉ failures then
17:         save child to the failures directory
18:         failures ← failures ∪ {failure}
19:     end if
20:     population ← update(population, feedback, child)
21: end while
```

2.2 Crossover

Crossover (sometimes also referred to as recombination or splicing) is an evolutionary operator that produces new child inputs by combining multiple parent inputs with the goal of passing on desirable traits from the parents to the children [36]. Typically, crossover operators exchange segments from two parents between a number of "crossover points". For example, the crossover operator used by AFL and AFL++ is a one-point crossover — it combines two inputs by splicing them together at a randomly selected midpoint [16, 35]. Evolutionary search approaches commonly use one- or two-point crossover since performance may degrade as the number of crossover points increases [14, 15]. A crossover operator is most effective when it is able to recombine high-fitness, interesting subcomponents from separate parents into a single child [17, 36, 55].

2.3 Parametric Fuzzing

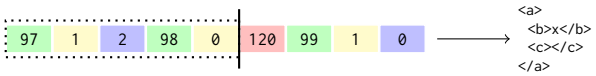
When a system under test has a highly constrained input structure, small modifications to an input are likely to produce invalid inputs preventing the fuzzer from exercising the core functionality of the system. Parametric fuzzing overcomes this limitation by using QuickCheck-style generators to achieve structural awareness [23, 38, 46]. The parametric fuzzer provides a means of splitting parts of fuzzer-created bit strings, known as "parameter sequences", into arbitrary, primitive-typed values. QuickCheck-style generators create complex input structures using these arbitrary values, thereby creating a "parametric generator" that maps parameter sequences to generated structures. For example, consider the `generate` method in Listing 1.

The method `nextByte` is provided by the parametric fuzzer; it consumes and returns the next byte of the parameter sequence. The `generate` method recursively creates an XML element. The call to `nextByte` on line 2 selects a tag name for the XML element. The value returned by the call to `nextByte` on line 4 determines whether the element should have child elements or text content. If the element has children, the call to `nextByte` on line 5 determines the number of children. Otherwise, the call to `nextByte` on line 9 selects the text content of the element.

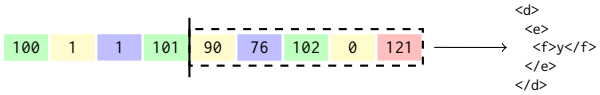
When the parameter sequence in Figure 1a is applied to the generator in Listing 1, the generator produces the string `"<a><b>x</b><c></c></a>"`. The parameter at index 0 is used to construct the root element's tag name, `'a'`. Next, the parameter at index 1 determines that the root element should have child elements, and the parameter at index 2 determines that there should be 2 children. The parameter at index 3 determines the first child's tag name, `'b'`. The parameter at index 4 determines that the first child should have text content, and the parameter at index 5 determines that the text content should be `'x'`. The parameter at index 6 determines the second child's tag name, `'c'`. The parameter at index 7 determines that the second child should have children. Finally, the parameter at index 8 determines that the second child should have 0 children.

```
1 public String generate() {
2   char n = (char) nextByte();
3   String c = "";
4   if (nextByte() > 0) {
5     int x = nextByte() % 5;
6     for (int i = 0; i < x; i++) {
7       c += generate();
8     }
9   } else {c += (char) nextByte();}
10  return "<" + n + ">" + c +
11    "</" + n + ">";
12 }
```

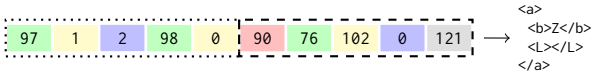
Listing 1: A simple XML document generator.



(a) Parent A. The solid line marks the crossover point. The dotted line encloses the prefix contributed to the child produced from the crossover operation.



(b) Parent B. The solid line marks the crossover point. The dashed line encloses the suffix contributed to the child produced from the crossover operation.



(c) Child produced by performing one-point crossover on Parents A and B. The dotted and dashed lines enclose the portions of the child's parameter sequence that were transferred from Parents A and B, respectively.

Figure 1: A generator (left) and associated inputs (right). For each input, we depict the bytes of the parameter sequence (left of the arrow) and the XML structure (right of the arrow) generated when the input is applied to the generator. Each parameter is colored based on how it used by the generator. Parameters used to construct a value returned by the call to `nextByte` on lines 2, 4, 5, and 9 are colored green, yellow, blue, and red, respectively. Unused parameters are colored gray. Whitespace has been added to the XML structures for readability.

Following a similar process, the generator produces the string `"<d><e><f>y</f></e></d>"` from the parameter sequence in Figure 1b.

Notice that the effect of each parameter depends upon how it is used by the generator. For example, the parameter at index 5 of Figure 1a is used by the generator to create the text content `'x'` for the first child of the root element. Whereas, the parameter at index 5 of Figure 1b is used by the generator to determine that the first child of the root element should have 1 child.

Since a parameter sequence is a bit string, it can be perturbed using generic mutation and crossover operators to create a new child sequence. Regardless of how these operators change the parent, the structure generated from the child sequence will still conform to any constraints imposed by the generator. For instance, the generator in Listing 1 will always create opening and closing tags. This allows the parametric fuzzer to produce valid inputs of various types without the need for type-specific operators. However, since the effect of a parameter value depends on the context in which it is used, unmodified parameters in the child sequence may be interpreted differently than they were for the parent sequence. This can limit the effectiveness of traditional crossover operators because naively chosen crossover points are likely to cause the parameters from one parent to be placed into a position in the other parent that corresponds to an entirely different context.

Consider a one-point crossover of the inputs in Figures 1a and 1b that produces the child parameter sequence displayed Figure 1c. This child parameter sequence generates the string `"<a><b>Z</b><L></L></a>"` when applied to the generator in Listing 1. Even though the child parameter sequence was constructed from part of the sequence in Figure 1b, the structure generated from the child does not resemble the structure generated for Figure 1b, because the portion of the sequence in Figure 1b that was transferred to the child was interpreted in a different context. For example, the parameter at index 5 of input sequence in Figure 1b, 76, was originally used by the call to `nextByte` on line 4. In this context, the value 76 meant that the element should have children. However, in the child parameter sequence, the parameter value 76 was used by the call to `nextByte` on line 2. In this context, the value 76 meant that the element should have the tag name `'L'` (ASCII character 76).

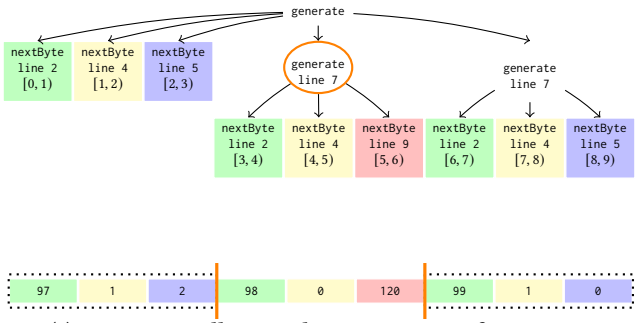
These context changes limit the number of crossover points that produce children that inherit advantageous traits from both parents — negatively impacting the heritability of traditional crossover operators. This effect is even more pronounced as the length of parameter sequences and the complexity of generators increase.

3 APPROACH

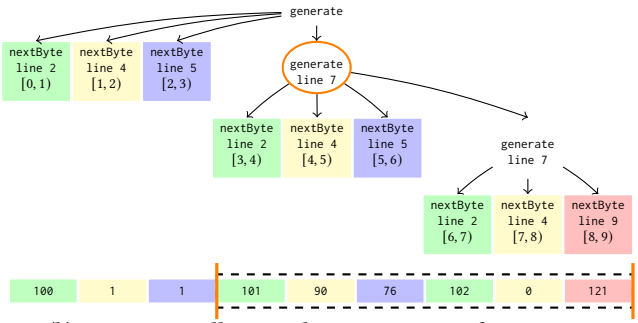
Our approach to crossover in parametric fuzzing, *linked crossover*, leverages information about the dynamic execution behavior of parametric generators to intelligently select crossover points. Linked crossover aims to identify and exchange portions of parameter sequences that are interpreted similarly by the parametric generator. These subsequences are identified using "parametric call trees". A parametric call tree records caller-callee relationships between method calls and the portion of the parameter sequence that was used by each method call. Linked crossover is a variant of two-point crossover — two crossover points are chosen for each parent and the values between those points are swapped between the parents. Unlike traditional two-point crossover, which chooses crossover points at random [15], linked crossover computes crossover points based on the parametric call trees of the parent inputs. This links the choice of crossover points to the parametric generator's execution behavior, thereby preserving logical boundaries in the input and increasing the chance that the crossover produces a child that inherits traits from both parents.

3.1 Parametric Call Tree

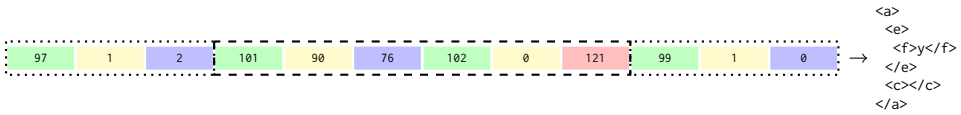
The parametric call tree for a parameter sequence represents the execution of the `generate` method, the method responsible for constructing arguments for the fuzzing target from a parameter sequence using one or more parametric generators, when `generate` is supplied with the parameter sequence. The parametric call tree



(a) Parametric call tree and crossover points for Parent A.



(b) Parametric call tree and crossover points for Parent B.



(c) Child sequence (left) and the XML structure generated when the child is applied to the generator in Listing 1 (right).

Figure 2: Parent parametric call trees (top), crossover points (middle), and the child parameter sequence (bottom) for a linked crossover. Parametric call trees are depicted for the generator executions produced when applying the parent parameter sequences in Figures 1a and 1b to the generator in Listing 1. Each vertex contains the name of the method called and the line on which it was called. Parameters requests consumed by calls to `nextByte` on lines 2, 4, 5, and 9 are colored green, yellow, blue, and red, respectively. The third line of each parameter request indicates the parameter sequence interval consumed by the request. Arrows represent caller-callee relationships. Below each tree, we show the parent parameter sequence and the crossover points linked to the vertex circled in the tree. Crossover points are marked with solid orange lines. Below the parents, we show the child parameter sequence produced from crossing over the parents at the marked points. The dotted and dashed lines enclose the portions of the child that were transferred from Parents A and B, respectively.

consists of a set of vertices representing method calls and edges representing caller-callee relationships. Each vertex has exactly one parent, its caller, except the root of the tree, which has no parent and represents the initial call to `generate`. Each vertex v has zero or more child vertices representing method calls made directly by v, *i.e.,* its callees. A vertex v is a descendant of a vertex u if the simple path from the root of the tree to v contains u; every vertex is a descendant of itself. Therefore, a vertex v is a descendant of a vertex u if the method call represented by v was made during the execution of the method call represented by u or $v = u$.

Vertices corresponding to calls to methods provided by the parametric fuzzer that directly consume bytes from the parameter sequence (*e.g.,* the `nextByte` method used in Listing 1) are annotated with the interval of the parameter sequence that was consumed by the call. We refer to these annotated vertices as "parameter requests". A parameter request has no children by construction. A method call is represented in the parametric call tree if and only if it is a parameter request or at least one parameter request occurred during the execution of that method call. Therefore, for every vertex v in a parametric call tree, there exists some parameter request u that is a descendant of v.

In theory, the parametric call tree could be defined per thread of execution. However, since parametric generators are single-threaded (regardless of whether the application under test is multi-threaded), parameter requests only occur in a single thread. Thus, we will discuss only a single parametric call tree for each input parameter sequence.

As an example, consider the generator in Listing 1 and the parametric input depicted in Figure 1a. The parametric call tree for the execution of `generate` induced by the parameter sequence in Figure 1a is shown in Figure 2a. The root vertex represents the call to `generate`. The leftmost child of the root represents the first call to `nextByte` on line 2. This vertex is a parameter request and is associated with the interval $[0, 1)$ because the first call to `nextByte` consumed the first byte of the parameter sequence. The next two leftmost children of the root vertex represent the calls to `nextByte` on lines 4 and 5. The two rightmost children of the root vertex represent recursive calls to `generate` made on line 7. Each of these two vertices has three children, each representing a call to `nextByte`.

3.2 Linked Crossover

Computing Crossover Points. When performing linked crossover, the crossover points for a parent input are computed based on a vertex selected from the parent's parametric call tree using Algorithm 2. The computed crossover points split the parent input before and after the portion of the parameter sequence that was used by the parameter requests made during the execution of the method call represented by the vertex. This preserves boundaries corresponding to method calls in the input increasing the chance that high-fitness subsequences in parent inputs appear in their children.

For example, consider the circled vertex in Figure 2a. There are three parameter requests that are a descendant of this vertex. The union of the intervals associated with these requests is $[3, 6)$ corresponding to the crossover points depicted in Figure 2a.

Selecting Vertices. In order to leverage vertex-based crossover points to combine inputs from a population, linked crossover begins by selecting two vertices: a "recipient" and a "donor". Given a parent

Algorithm 2 Computing crossover points for a vertex.

Input: parametric call tree vertex v
Output: a pair of crossover points
1: $D \leftarrow$ the set of parameter requests that are a descendant of v
2: $S \leftarrow \{\}$
3: **for each** $d \in D$ **do**
4: $S \leftarrow S \cup$ interval of parameter sequence consumed by d
5: **end for**
6: **return** $min(S), max(S) + 1$

input, referred to as the "primary" parent, linked crossover begins by selecting a recipient vertex at random from eligible vertices in the primary parent's parametric call tree. A vertex v is eligible to be a recipient if the following is true:

(1) There does not exist some vertex u, such that the set of parameter requests descended from v is equal to those descended from u and the method call represented by u happened before the method call represented by v.
(2) v is not the root of the parametric call tree.
(3) There are at least two different parameter requests that are a descendant of v.

The first criterion is violated if a vertex does not have a sibling. A vertex does not have a sibling if every parameter request that occurred during the caller of the method represented by the vertex occurred during execution of the method represented by the vertex. The first criterion ensures that there is only one eligible vertex corresponding to each distinct pair of crossover points. If two vertices, u and v, have the same set of descendant parameter requests, then the crossover points for v and u are the same. Therefore, when two or more vertices have the same set of descendant parameter requests, only the vertex corresponding to the earliest method call is eligible to be a recipient. The second criterion guards against producing a child that is overly *dissimilar* to the primary parent: if v is the root of the parametric call tree, then selecting v as the recipient will replace the entire parameter sequence for the primary parent. The third criterion is a heuristic that guards against producing a child that is overly *similar* to the primary parent: if only one parameter request is a descendant of v, then selecting v as the recipient will replace only a small portion of the parameter sequence for the primary parent. Future work could explore other heuristics for selecting recipient vertices.

Next, linked crossover selects a secondary parent at random from the set of eligible members of the population. A member of the population is eligible to be the secondary parent if its call tree contains at least one eligible vertex. Every vertex that represents a call to the same method as the recipient is eligible to act as the donor vertex, even vertices that do not satisfy the recipient eligibility requirements. This criterion increases the chance that the donated subsequence is interpreted similarly by the generator for the child as it was for the secondary parent. To reduce the performance impact of selecting secondary parents, the fuzzer can maintain a mapping from each method to the set of members of the current population that are eligible to act as the secondary parent for a linked crossover targeting a recipient vertex representing that method. Once a secondary parent is selected, a donor vertex is selected at random from eligible vertices in the secondary parent's parametric call tree.

Application. Algorithm 3 describes how to apply a linked crossover between a primary and secondary input based on a selected recipient vertex and donor vertex. Crossover points are computed based on the recipient and donor vertex, as described above. Standard two-point crossover is then performed replacing the portion of the primary parent's parameter sequence that lies between the crossover points computed for the recipient vertex with the portion of the secondary parent's parameter sequence that lies between the crossover points computed for the donor.

Algorithm 3 Applying a linked crossover.

Input: primary parent $x \leftarrow \langle x_0, x_1, \ldots, x_{n-1} \rangle$, secondary parent $y \leftarrow \langle y_0, y_1, \ldots, y_{m-1} \rangle$, recipient vertex r, donor vertex d
Output: child sequence
1: $i, j \leftarrow$ crossover points computed for r using Algorithm 2
2: $k, l \leftarrow$ crossover points computed for d using Algorithm 2
3: **return** $\langle x_0, x_1, \ldots, x_{i-1} \rangle + \langle y_k, y_{k+1}, \ldots, y_{l-1} \rangle + \langle x_j, x_{j+1}, \ldots, x_{n-1} \rangle$

Figure 2 depicts a linked crossover between the two inputs in Figure 1. The recipient vertex is the call to the method `generate` circled in orange in Figure 2a. The donor vertex is the call to the method `generate` circled in orange in Figure 2b. The union of the interval of parameter requests descended from the recipient vertex is $[3, 6)$ and from the donor vertex is $[3, 9)$. These intervals correspond to the crossover points marked with orange lines in Figures 2a and 2b. The portion of the sequence in Figure 2a between the marked crossover points is replaced with the portion of the sequence in Figure 2b between the marked crossover points producing the child sequence depicted in Figure 2c.

When applying multiple linked crossover operations to the same primary parent input additional considerations must be made because an operation may shift subsequent portions of the input if the size of the donated subsequence is not equal to the size of the replaced subsequence. Additionally, if two operations impact non-disjoint intervals of the primary parent's parameter sequence, only one of the operations can be applied because the same parameter should only be replaced once.

When applying multiple linked crossover operations on the same primary parent, begin by sorting the operations into non-increasing order by the start of the interval they impact (as determined by the recipient vertex of the application). Then, process each operation in order. If an operation targets an interval that is non-disjoint with an interval targeted by an operation that has already been applied, skip it. Otherwise, apply the operation as normal replacing the portion of the primary parent within the interval targeted by the operation with the operation's donated subsequence.

Applying the linked crossover operations in non-increasing order by the start of the interval they impact ensures that parameters in positions before the start of the interval targeted by the last operation applied remain in their original positions. When a crossover operation is about to be applied, the interval that it targets must be disjoint with the interval targeted by the last operation applied, otherwise the operation would have been skipped. Furthermore,

the start of the interval targeted by the operation cannot be greater than the start of the interval targeted by the last operation applied. Therefore, when an operation is applied, the interval that it targets must start and end before the start of the interval targeted by the last operation applied. Thus, the operation can be applied normally without adjusting the replacement interval.

4 IMPLEMENTATION

Although we believe that linked crossover is suitable for many languages, we implemented linked crossover as part of Zeugma, a new parametric fuzzer for Java. For the sake of simplicity, we chose to use branch coverage feedback for Zeugma. However, linked crossover could be used with other forms of feedback such as Padhye et al. [46]'s input-validity feedback. Zeugma collects branch coverage and method call information using the ASM instrumentation and analysis framework to rewrite Java bytecode [44]. Like JQF [45] (the parametric fuzzing framework used to create Zest, BeDivFuzz, and RLCheck), Zeugma is implemented on top of junit-quickcheck [19], a property-testing library inspired by QuickCheck [10]. junit-quickcheck leverages user-defined generators to create random test inputs. These generators use a high-level API provided by junit-quickcheck to create arbitrary values of common types. Zeugma integrates into junit-quickcheck by using fuzzer-derived parameter sequences to determine these arbitrary values.

Updating the Population. Zeugma implements the generic fuzzing algorithm described in Section 2.1. Branch coverage feedback is used to determine which inputs should be included in the population. For each branch that has been covered by at least one input, Zeugma tracks the shortest input that covered that branch. The set of tracked inputs form the population.

Modifying Inputs. When creating a child input, Zeugma selects the primary parent from the population at random. Then, Zeugma chooses the total number of mutation and crossover operations to apply to the primary parent. This number is chosen from a shifted geometric distribution with a success probability of 0.25 (corresponding to an expected value of four). This value can be fine-tuned; preliminary experiments that we conducted suggested that a probability of 0.25 was an effective choice across all subjects. Zeugma uses the same replacement-based mutation operator described by Padhye et al. [46] with a mean mutation length of eight. The mean mutation length can be fine-tuned; preliminary experiments that we conducted suggested that eight was an effective choice across all subjects.

Zeugma can be configured to use mutation only, to use traditional one-point crossover, to use traditional two-point crossover, or to use our novel linked crossover. We describe our evaluation of these options in Section 6. For one- and two-point crossover, the second parent is selected at random from the population, and crossover points are selected at random. Linked crossover is performed as described in Section 3.2.

If Zeugma has been configured to use mutation only, then all the operations are mutation. Otherwise, Zeugma chooses between mutation and crossover at random with an equal likelihood of selecting either option. If Zeugma has been configured to use linked crossover, then all linked crossover operations are applied first in the manner described in Section 3.2, then the mutation operations are applied. Otherwise, operations are applied in the order they are selected.

Building the Parametric Call Tree. Zeugma uses bytecode instrumentation to build parametric call trees allowing linked crossover to work on unmodified junit-quickcheck generators. Zeugma adds code at the start of methods that records that the method was entered and before method returns that records that the method was exited. Zeugma's integration with junit-quickcheck records when a portion of the parameter sequence is consumed. These recorded messages are ignored by Zeugma unless it is actively building a parametric call tree.

Parametric call trees are only needed when Zeugma is configured to use linked crossover and for inputs that will be saved to the population. Hence, before Zeugma saves an input to the population, it re-executes the `generate` method with the input and observes messages recorded about method entries, method exits, and parameter consumptions. A stack is used to track the call stack. When a method is entered, a new vertex is created for the method call and pushed onto the stack. When a parameter is consumed, the index of that parameter is then associated with the vertex at the top of the stack and the top vertex is marked as a parameter request. When a method is exited, the top vertex is popped off of the stack. If the popped vertex is not associated with the index of at least one parameter, and it has no children, then the vertex is not a parameter request and no parameter requests occurred during the execution of the method call represented by the vertex. As noted in Section 3.1, a method call is included in the parametric call tree only if it is a parameter request or at least one parameter request occurred during the execution of that method call. Therefore, the popped vertex is discarded. If the popped vertex is not discarded, then the vertex on the top of the stack is marked as the parent of the popped vertex. If there is no vertex on the top of the stack, *i.e.,* the stack is empty, then the popped vertex is recorded as the root of the tree.

5 LIMITATIONS

Linked crossover is a heuristic approach; its efficacy is dependent on the structure of the generators. If the generators are not split into methods or the methods do not correspond to logical boundaries, then linked crossover will be ineffective. This limitation only applies to the structure of the generators and not the entire system under test. However, we do not believe that this is a significant limitation, as best practices for writing QuickCheck-style generators rely on composition. Claessen and Hughes [10] explain that combinators can be used to combine simple generators into complex generators. This compositional style results in method calls that are responsible for creating a single subcomponent, and, therefore, likely correspond to reasonable logical boundaries. Linked crossover's dependency on methods could be mitigated by using additional dynamic execution information or techniques such as method call sites, dynamic slices, and dynamic information flows. This is an interesting direction for future research.

6 EVALUATION

For our evaluation, we examined linked crossover's impact on overall fuzzer performance and its ability to produce children that preserve desirable traits from their parents — a property referred to by Raidl and Gottlieb [48] as heritability. We created two novel heritability metrics for evolutionary fuzzing, *hybrid proportion* and *inheritance rate*, which we describe in Section 6.1.1. We also examined the impact of linked crossover on overall fuzzer performance using traditional metrics. Our evaluation of linked crossover focused on answering the following research questions:

RQ1: How does linked crossover compare to other crossover operators with respect to heritability?

RQ2: How does Zeugma with linked crossover's ability to discover coverage-revealing inputs compare to state-of-the-art parametric fuzzers?

RQ3: How does Zeugma with linked crossover's ability to detect defects compare to state-of-the-art parametric fuzzers?

6.1 Methodology

We evaluated Zeugma on benchmark suite of seven real-world Java projects consisting of the five subjects used by Padhye et al. [46] in their evaluation of Zest (Ant, BCEL, Closure, Maven, and Rhino) and the two additional subjects used by Nguyen and Grunske [38] in their evaluation of BeDivFuzz (Nashorn and Tomcat). We list these subjects in Table 1. We used the latest stable release of each subject available in the Maven Central Repository. Minor modifications were made to the fuzzing targets used by Padhye et al. [46] and Nguyen and Grunske [38] to ensure compatibility with the newer subject versions. We used the junit-quickcheck generators included with JQF (version 2.0) [50] for XML, JavaScript, and Java classes. We changed the configuration for the XML generator to increase the maximum depth of generated XML trees to ten as Kukucka et al. [23] found that deeper trees were necessary to exercise certain functionality in Maven. No changes were made to the generator itself.

In order to compare linked crossover against other crossover operators, we created four variants of Zeugma. The first variant, Zeugma-X, does not use crossover at all. The other variants, Zeugma-Link, Zeugma-1PT, and Zeugma-2PT, use linked, one-point and, two-point crossover, respectively. These variants differ from each other only with respect to the application of crossover as described in Section 4.

Table 1: Evaluation Subjects. For each subject, we list the project name and version (Project), the format of the input (Format), and the number of branches as reported by JaCoCo (Branches).

Project		Format	Branches
Apache Ant (1.10.13)	[1]	XML	24626
Apache BCEL (6.7.0)	[4]	Java class	5975
Google Closure (v20230502)	[13]	JavaScript	129376
Apache Maven (3.9.2)	[5]	XML	14886
OpenJDK Nashorn (11.0.19)	[39]	JavaScript	28191
Mozilla Rhino (1.7.14)	[33]	JavaScript	26690
Apache Tomcat (10.1.9)	[6]	XML	39020

6.1.1 RQ1: Heritability. In an evolutionary search, an effective crossover operator produces children that preserve desirable traits from their parents — a property referred to by Raidl and Gottlieb [48] as heritability. For an evolutionary fuzzer, the primary trait of interest for an input is its ability to cover program features (typically branches or statements). We propose two coverage-based heritability metrics for evolutionary fuzzing: *inheritance rate* and *hybrid proportion*. Inheritance rate considers the percentage of program features covered by at least one of an input's parents that were also covered by the input for a typical input produced by a crossover operator. Hybrid proportion measures the likelihood that a crossover operator produces a child that covers at least one feature exclusively covered by each of its parents.

Inheritance rate and hybrid proportion aim to measure commonalities between a child and its parents — they do not try to measure whether the child covers new program features. Although additional coverage is generally positive in fuzzing, it is not necessarily indicative of a high-quality crossover. The additional coverage could represent an undesirable deviation from the parents' behavior, or it could be a positive effect of combining parts of the parents' behavior. Therefore, program features covered by the child but not its parents are neither penalized nor rewarded when computing inheritance rates and hybrid proportions.

Consider a child c produced by applying a crossover operator to parents p_1 and p_2. Let X be a set of "common" features — program features that are covered by a high percentage of random inputs. Common features are excluded when computing inheritance rate and hybrid proportion because covering a common feature does not necessarily represent a unique, desirable property of a particular input. Let P_1, P_2, and C be the set of program features not in X covered by p_1, p_2, and c, respectively. We define the *inheritance rate*, denoted IR, of a crossover as the percentage of program features covered by either parent that were also covered by the child:

$$\mathrm{IR}(P_1, P_2, C) = \frac{|(P_1 \cup P_2) \cap C|}{|P_1 \cup P_2|}$$

We say that a crossover is a *hybrid*, denoted HY, if the child covers at least one feature that is covered by the first parent but not the second parent, and the child covers at least one feature that is covered by the second parent but not the first parent:

$$\begin{aligned}\mathrm{HY}(P_1, P_2, C) = (\exists x \in C : x \in P_1 \wedge x \notin P_2) \\ \wedge (\exists y \in C : y \in P_2 \wedge y \notin P_1)\end{aligned}$$

These metrics are extended to the operator itself by considering the distribution of inheritance rates and the proportion of children that are hybrids for some sample of parents.

In order to collect a representative sample of parent inputs, we performed twenty fuzzing campaigns using Zeugma-X on the subjects listed in Table 1 and recorded the state of the corpus after five minutes. We chose to use the state of the corpus after five minutes because we found that, on average, over half of the inputs saved to corpus after three hours were saved in the first five minutes (mean = 57.8%, median = 61.8%, minimum = 29.8%, maximum = 78.2%). We collected branch coverage using Zeugma's instrumentation, and considered all coverage including system classes. To ensure that non-repeatable coverage due to class loading did not impact the results, if a class was loaded during the execution of an input, the

input was re-executed, and the coverage recording from the second execution was used.We identified the set of common features, X, by executing 1,000 random inputs for each subject. Any branch that was covered by a majority of these random inputs was marked as common.

To compute the heritability metrics, we sampled 1,000 pairs of parents for each subject. Each sample was selected by choosing two parents at random from a randomly selected corpus produced for the subject. Samples were re-selected until both parents covered at least one branch not in the common feature set and not covered by the other parent to ensure that it was possible to produce a hybrid child from the pairing. For each sample, we produced one child for each of the three crossover operators — linked, one point and two point — and recorded whether the child was a hybrid and its inheritance rate.

6.1.2 RQ2 and RQ3: Coverage and Defects. Our second and third research questions evaluate the impact of linked crossover on branch coverage and defect detection ability. In addition to the three variants of Zeugma without linked crossover, we also compared our approach against Zest [46], BeDivFuzz [38], and RLCheck [49]. We used the latest releases at time of writing of Zest (version 2.0) and BeDivFuzz (commit c06eaca) which include improvements to the coverage instrumentation. For BeDivFuzz, we evaluated both of the configurations described by Nguyen and Grunske [38]: BeDiv-Struct and BeDiv-Simple. Because our reported values are based only on saved inputs, we modified BeDivFuzz to ensure that all inputs that reveal new coverage are saved — not just "valid" inputs. For the same reason, we choose to use the "grey-box" version of RLCheck because the grey-box version saves coverage-revealing inputs to a corpus. We did not compare Zeugma against Confetti [23] because Confetti only supports Java version 8, and the latest release of Zest requires Java version 9 or greater.

BeDivFuzz and RLCheck cannot use junit-quickcheck generators out of the box; they require the generators to be manually modified. For the XML and JavaScript generators, we used the generators created by Nguyen and Grunske [38] and Reddy et al. [49] to evaluate BeDivFuzz and RLCheck, respectively. Neither Nguyen and Grunske [38]'s evaluation of BeDivFuzz nor Reddy et al. [49] evaluation of RLCheck included a Java class generator. Therefore, we created a modified version of the Java class generator included with JQF for BeDivFuzz. Unfortunately, the documentation for RLCheck did not provide sufficient detail for us to create a modified Java class generator for RLCheck; therefore, we do not include results for RLCheck on BCEL.

Following best practices suggested by Metzman et al. [34], we used an independent code coverage metric — branch coverage collected with JaCoCo (version 0.8.7) [37]. In order to calculate branch coverage, we reran inputs saved by the fuzzer in a JaCoCo-instrumented Java Virtual Machine (JVM) after the campaign finished. Coverage was measured only in application classes (those found in the JAR files associated with the subject). For Nashorn, we further limited coverage to only include classes related to Nashorn, those with the package prefix `jdk.nashorn`. Nashorn is part of the Java Class Library (JCL) and including all JCL coverage would bias results in favor of fuzzers that heavily depend on parts the JCL (*e.g.,* `java.lang.String` and `java.util.HashMap`) inflating their coverage.

Table 2: Heritability Metrics. For each crossover operator, we report the proportion of samples that were hybrids (**HY**) and the median inheritance rate (**IR**) on each subject. The largest value for each metric on each subject is highlighted in blue. Values that differ significantly from that of linked crossover are colored red.

	Linked		One Point		Two Point	
Subject	HY	IR	HY	IR	HY	IR
Ant	0.561	0.923	0.459	0.124	0.493	0.069
BCEL	0.283	0.512	0.660	0.347	0.756	0.286
Closure	0.742	0.717	0.661	0.101	0.712	0.094
Maven	0.446	0.589	0.404	0.497	0.399	0.453
Nashorn	0.622	0.646	0.548	0.117	0.591	0.132
Rhino	0.611	0.502	0.599	0.263	0.643	0.255
Tomcat	0.322	0.775	0.350	0.276	0.328	0.279

In order to measure defect detection ability, we collected the failures detected for each campaign by rerunning inputs saved by the fuzzer in a new JVM after the campaign finished. If a saved input induced a failure, we recorded the type (*e.g.,* `java.lang.RuntimeException`) and stack trace of the failure induced by the input. Failures with the same type and top five stack frames were initially marked as the same failure. We then manually inspected the set of distinct failures to map the failures to a set of unique defects. All of the identified defects were reported and confirmed by a developer for the associated project.

Twenty trials were conducted for each fuzzer on each subject in accordance with current best practices [22]. Each campaign was performed on its own virtual machine with four 2.6 GHz AMD EPYC 7H12 vCPUs, with 16 GB of RAM, running Ubuntu 20.04.3, and using the Oracle Java Development Kit (JDK) version 11.0.19. No seeds were provided for any subject. The original dictionaries created by Padhye et al. [46] and Nguyen and Grunske [38] were used for the XML subjects. Similar to the evaluation performed by Padhye et al. [46] and Kukucka et al. [23], generators were limited to producing inputs that used 10,240 or less raw input bytes.

Because parametric fuzzing is typically used for property-based testing, the effectiveness a parametric fuzzer on relatively short campaigns is of particular interest [38, 49]. However, longer campaigns may be more indicative of general performance trends [22]. Therefore, we chose to evaluate the effectiveness of linked crossover on both short (five minute) campaigns like Reddy et al. [49] and long (three hour) campaigns like Padhye et al. [46].

6.2 RQ1: Heritability

Table 2 summarizes the results of our heritability experiment. We performed pairwise comparisons of the inheritance rates and hybrid proportions measured on each subject for the different crossover operators using two-tailed Mann-Whitney U tests and Fisher's exact tests, respectively. Following current best practices as described by Arcuri and Briand [7], a base significance level of 0.05 was adjusted for three comparisons resulting in a Bonferroni-corrected significance level of $\frac{0.05}{3} = 0.0167$ per test. Inheritance rates for

linked crossover were statistically significantly greater than for one- and two-point crossover on all subjects. Linked crossover had a significantly greater hybrid proportion compared to one-point crossover on Ant, Closure, and Nashorn; and compared to two-point crossover on Ant. However, linked crossover had a significantly lower hybrid proportion compared to one- and two-crossover on BCEL. All other differences between linked crossover and the other operators were not significant. Full results of the pairwise comparisons are included in the supplemental materials for this paper. Overall, linked crossover produced children that inherited more desirable traits from their parents (*i.e.,* covered a higher percentage of the branches covered by their parents) while still combining traits from both parents.

6.3 RQ2: Coverage

Figure 3 and Table 3 summarize the results of our coverage experiment. Figure 3 shows median, minimum, and maximum branch coverage across the twenty trials over time for each fuzzer on each subject. Table 3 shows median coverage values for each fuzzer after five minutes (short duration) and three hours (long duration). On each subject, we performed pairwise comparisons of the branch coverage for short and long campaigns for the different fuzzers using Mann-Whitney U tests. A base significance level of 0.05 was adjusted for 28 comparisons resulting in a Bonferroni-corrected significance level of $\frac{0.05}{28} = 0.00179$ per test except on BCEL. For BCEL, the significance level was adjusted for 21 comparisons (due to the exclusion of RLCheck) resulting in a corrected significance level $\frac{0.05}{21} \approx 0.00238$. Statistically significant differences between the branch coverage of Zeugma-Link and the other fuzzers are colored red in Table 3. We also used the Vargha-Delaney $\hat{A}_{12}$ statistic [53] to quantify effect sizes for these comparisons. The full results of these tests are included in our supplemental materials.

Zeugma-Link had the highest median coverage on all subjects except Tomcat after five minutes and on all subjects except Ant after three hours. On the three JavaScript subjects (Closure, Nashorn, and Rhino), Zeugma-Link's branch coverage was significantly greater than that of all other fuzzers on both long and short campaigns, except Zeugma-X on five-minute Closure campaigns. On the only Java class subject (BCEL), Zeugma-Link outperformed BeDiv-Simple, BeDiv-Struct, RLCheck, and Zeugma-2PT on long and short campaigns. However, Zeugma-Link only performed significantly better than Zest, Zeugma, and Zeugma-1PT on long BCEL campaigns. We carefully examined this fuzzing target and believe that, in order to achieve further improvements in coverage, the generator should be improved to be more likely to generate valid Java method bodies. Results on the three XML subjects (Ant, Maven, and Tomcat) were more mixed. As depicted in Figure 3, coverage for most of the fuzzers plateaued on these three subjects. Our analysis of these fuzzing targets revealed that limited coverage is reachable from the drivers for these subjects. Zeugma-Link performed as well as or better than the other fuzzers on all the XML subjects except on long Ant campaigns where Zeugma-Link's branch coverage was significantly less than that of Zest. For all subjects, we found that the effect size was large ($\hat{A}_{12} \geq 0.71$) for all comparisons in which the performance of a baseline fuzzer differed significantly from that of Zeugma-Link.

BeDivFuzz, Zest, and RLCheck performed notably poorly on Nashorn. These fuzzers are all built using JQF which does not add and cannot be configured to add coverage instrumentation to classes with the package prefix `jdk`. Since the classes related to Nashorn are found in `jdk.nashorn`, the JQF-based fuzzers did not receive critical coverage feedback for Nashorn. Given that our primary goal is to evaluate the efficacy of linked crossover (as opposed to comparing all variants of Zeugma against BeDivFuzz, Zest, and RLCheck), we did not find it necessary to make the invasive changes to JQF necessary to collect coverage in these packages.

In general, linked crossover was demonstrably effective at discovering coverage-revealing inputs in both long and short campaigns. It consistently performed as well as or better than other forms of crossover and against state-of-the-art parametric fuzzers. Linked crossover was generally more effective on subjects using the JavaScript or Java class generator, possibly indicating that its efficacy may be impacted by either the nature of the generator or the input type itself. However, this could also be a product of driver limitations for the XML fuzzing targets. Future work may study how to measure and improve the quality of fuzzing targets.

6.4 RQ3: Defects

Across all the campaigns, a total of twelve unique defects were detected: two in BCEL (B0–B1), two in Closure (C0–C1), three in Nashorn (N0–N2), and five in Rhino (R0–R4). Table 4 lists the percentage of campaigns for each fuzzer in which each of these defects was discovered (the detection rate) within the first five minutes and after the full three hours of the campaign. For each defect, we performed pairwise comparisons of the detection rate for short and long campaigns for the different fuzzers using Fisher's exact tests with the same Bonferroni-adjusted significance levels used in Section 6.3. Statistically significant differences between Zeugma-Link and other fuzzers are colored red in Table 4. Full results of these tests are included in our supplemental materials.

Zeugma-Link had the highest detection rate for nine of the twelve defects within the first five minutes (short campaigns) and eight out of the twelve defects after the full three hours (long campaigns) of the campaign. This suggests that linked crossover may positively impact a fuzzer's ability to detect defects. For the short campaigns, the Fisher's exact tests indicated that most of the differences in detection rate between the fuzzers were not significant. For long campaigns, Zeugma-Link's detection rate for five defects (N0, N2, R0, R2, R4) was significantly higher than that of BeDiv-Simple, BeDiv-Struct, RLCheck, and Zest. Interestingly, Zeugma-Link's detection rate never differed significantly from that of Zeugma-X, although, in some cases, it was superior to that of Zeugma-1PT and Zeugma-2PT.

6.5 Threats to Validity

We evaluated linked crossover on only seven subjects. These subject may not be representative of all programs. However, these subjects are all mature, well-established projects. Furthermore, we included all the Java subjects evaluated by Padhye et al. [46], Reddy et al. [49], Kukucka et al. [23], or Nguyen and Grunske [38].

Our evaluation featured generators for only three different input types: JavaScript, XML, and Java class. We did not evaluate the

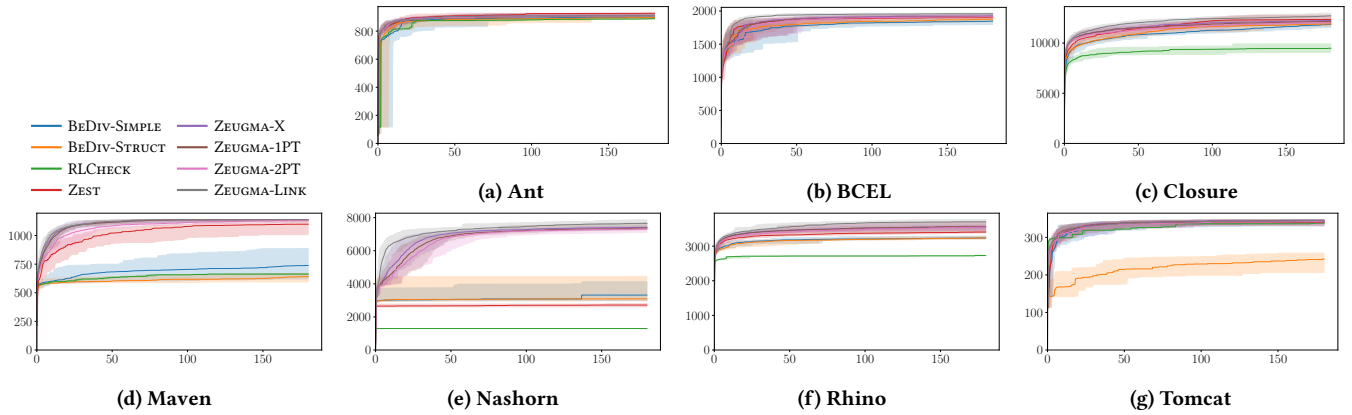


Figure 3: Branch Coverage Over Time. Each x-axis is time in minutes and each y-axis is the number of covered branches. Each plot depicts the median number of covered branches (solid line) and the range of covered branches (filled area) across the 20 trials over the duration of the fuzzing campaign for each of the fuzzers.

Table 3: Branch Coverage. For each fuzzer, we report the median branch coverage in application classes for each subject across 20 fuzzing campaigns after five minutes (5M) and three hours (3H). The largest median or medians (in the case of a tie) for each time and subject is highlighted in blue. Branch coverage values that differ significantly from Zeugma-Link's are colored red.

Fuzzer \ Subject	Ant		BCEL		Closure		Maven		Nashorn		Rhino		Tomcat	
	5M	3H	5M	3H	5M	3H	5M	3H	5M	3H	5M	3H	5M	3H
BeDiv-Simple	755.0	899.0	1435.5	1846.5	9328.0	11863.5	590.5	738.0	3008.5	3319.5	2952.0	3235.5	274.5	341.0
BeDiv-Struct	786.5	896.5	1412.5	1876.5	9336.5	11904.5	578.5	641.5	2993.5	3092.0	2915.5	3237.0	161.0	242.5
RLCheck	769.0	889.0	—	—	8262.5	9480.5	579.0	663.0	1298.0	1298.0	2627.0	2730.0	299.0	338.0
Zest	820.0	927.0	1516.5	1909.5	9782.5	12352.0	778.5	1098.5	2654.5	2717.0	3108.5	3408.0	295.0	340.0
Zeugma-X	835.5	911.0	1480.5	1927.0	10274.5	12251.0	873.0	1138.0	4259.0	7411.0	3169.0	3551.5	297.5	345.0
Zeugma-1PT	828.0	909.5	1489.0	1917.0	10237.5	12153.5	855.5	1138.0	4166.0	7369.5	3169.0	3572.5	294.0	344.0
Zeugma-2PT	819.5	910.0	1472.0	1915.0	10284.0	12038.0	797.0	1134.0	3962.5	7310.0	3143.0	3549.0	291.0	341.5
Zeugma-Link	845.5	911.5	1541.5	1959.0	10395.5	12709.0	906.0	1138.0	5568.5	7654.0	3233.5	3703.0	295.5	345.0

Table 4: Defect Detection Rates. For each fuzzer, we report the defect detection rate of each discovered defect across 20 fuzzing campaigns after five minutes (5M) and three hours (3H). The largest detection rate or rates (in the case of a tie) for each time and defect is highlighted in blue. Detection rates that differ significantly from Zeugma-Link's are colored red.

Fuzzer \ Defect	B0 [2]		B1 [3]		C0 [11]		C1 [12]		N0 [42]		N1 [40]		N2 [41]		R0 [31]		R1 [30]		R2 [28]		R3 [29]		R4 [32]	
	5M	3H	5M	3H	5M	3H	5M	3H	5M	3H	5M	3H	5M	3H	5M	3H	5M	3H	5M	3H	5M	3H	5M	3H
BeDiv-Simple	0.00	0.65	0.00	0.00	0.00	0.05	0.15	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.05	0.45	0.00	0.00	1.00	1.00	0.00	0.00
BeDiv-Struct	0.05	0.70	0.00	0.00	0.00	0.15	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.20	0.65	0.00	0.00	1.00	1.00	0.00	0.00
RLCheck	—	—	—	—	0.00	0.10	0.20	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00
Zest	0.00	1.00	0.00	0.00	0.00	0.40	0.10	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.10	0.05	0.95	0.00	0.20	1.00	1.00	0.00	0.05
Zeugma-X	0.05	1.00	0.00	0.00	0.05	0.25	0.60	1.00	0.00	0.75	0.00	0.00	0.00	0.05	0.00	0.75	0.15	0.80	0.00	0.95	1.00	1.00	0.00	0.90
Zeugma-1PT	0.00	1.00	0.00	0.10	0.00	0.40	0.50	1.00	0.00	0.30	0.00	0.00	0.00	0.00	0.00	0.70	0.10	0.75	0.00	0.85	1.00	1.00	0.00	0.85
Zeugma-2PT	0.25	1.00	0.00	0.10	0.00	0.40	0.80	1.00	0.00	0.45	0.00	0.05	0.00	0.05	0.00	0.60	0.10	0.90	0.00	0.70	1.00	1.00	0.00	0.85
Zeugma-Link	0.25	1.00	0.00	0.00	0.00	0.65	0.60	1.00	0.05	1.00	0.00	0.00	0.00	0.50	0.00	0.80	0.10	0.70	0.00	0.85	1.00	1.00	0.00	0.95

impact of generator quality or style on the effectiveness of linked crossover. Generator quality is likely to impact any generator-based technique — not just linked crossover. To avoid potential bias, we used the generators included with JQF without modification.

We used branch coverage as a metric for evaluating fuzzer effectiveness in Section 6.3. Coverage is only weakly correlated with defect detection ability [21]. However, as noted by Metzman et al. [34] the sparsity of bugs in programs makes it difficult to evaluate a fuzzer by analyzing detected defects alone. Therefore, we analyzed both branch coverage and defect detection rate.

7 RELATED WORK

Crossover in Unstructured Fuzzing. Generic crossover operators are often used by unstructured fuzzers, for example AFL [35], AFL++ [16], and libFuzzer [25]. As demonstrated in Section 6, generic crossover operators are less effective than linked crossover in parametric fuzzing. Lyu et al. [26] use Particle Swarm Optimization to find the optimal selection probability distribution of mutation and crossover operators.

Parametric Fuzzing. Padhye et al. [46] introduce the idea parametric fuzzing leveraging input-validity and coverage feedback to guide input generation. Reddy et al. [49] use reinforcement learning to guide generators to produce a diverse set of valid inputs. Nguyen and Grunske [38] perform structural-aware mutation which distinguishes between structure-changing and structure-preserving mutations to increase the behavioral diversity of inputs generated by a parametric fuzzer. Kukucka et al. [23] improve on parametric fuzzing by using "hinting", a form of intelligent mutation. Hints are identified based on comparisons against the input using concolic execution and taint tracking. Applied hints are saved to a global dictionary shared between inputs. Like linked crossover, the global dictionary allows high-fitness subcomponents to be transferred between inputs. Lampropoulos et al. [24] propose an alternative approach for guided and generator-based fuzzing that uses type-aware mutation operators instead of mutating a parameter sequence. These mutation operators are automatically synthesized based on input types, allowing the fuzzer to mutate inputs at the algebraic datatype. None of these approaches leverage crossover.

Specification-Based Fuzzing. Holler et al. [18] learn code fragments from a corpus of seed inputs using a context-free grammar for the input structure. Then, they modify inputs by randomly replacing fragments in the input with learned fragments of the same type. Wang et al. [54] and Aschermann et al. [8] incorporate coverage feedback into grammar-based fuzzing by employing tree-based mutation. One such mutation, "splicing mutation", proposed by Aschermann et al. [8] is a structured crossover that swaps subtrees between the derivation trees of two inputs. Srivastava and Payer [52] improve upon Aschermann et al. [8]'s coverage-guided, grammar-aware fuzzing approach by introducing larger, more "aggressive" mutations and restructuring input grammars' production rules to eliminate sampling bias. Pham et al. [47] use file format specifications to parse inputs into a virtual structure, a tree of file chunks. They then define structural mutation operators that operate on a file's virtual structure, such as, smart splicing — a form of structured crossover that transfers chunks between files.

Like most structured crossover operators, linked crossover aims to identify and exchange analogous, high-fitness subsequences between inputs. However, linked crossover does not require an input specification and is, therefore, capable of fuzzing inputs with constraints that cannot be represented by a particular type of specification. Instead, linked crossover leverages dynamic execution information to select crossover points.

Inferring Input Structure. You et al. [57] use a dynamic probing strategy to identify fields, regions of linked bytes, by observing the effect of applied mutations while fuzzing. Identified fields are then mutated using type-specific mutation strategies. You et al. [56] identify input validity checks on portions of the input and employ targeted mutation strategies to satisfy these checks. Blazytko et al. [9] infer structural properties of input formats over the course of a fuzzing campaign using code coverage feedback. Mathis et al. [27] use dynamic taint tracking infer lexical tokens and generate seed inputs for an input language.

Like these techniques, linked crossover infers properties of an input structure based on system behavior observed at runtime. However, unlike these techniques, linked crossover uses relationships between method calls and tracks input consumption to identify analogous regions of inputs.

8 CONCLUSION

This work demonstrates that crossover point selection can have a significant impact on overall fuzzer performance and that dynamic execution information can be effectively used to inform the selection of crossover points in evolutionary fuzzing. Linked crossover, our approach for using dynamic execution information to select crossover points, produced children that inherited more desirable traits from their parents than traditional one- and two-point crossover. Our evaluation of linked crossover's impact on fuzzer performance found that linked crossover was effective at discovering coverage-revealing inputs and defects in both long and short campaigns. Based on these results, we believe that linked crossover could potentially be adapted for use in unstructured fuzzing in cases where the input is read in a stream-like or piecewise manner by the application. The full results of the statistical tests that we conducted in our evaluation are available in our supplemental materials [20]. Zeugma's source code, our experimental scripts, and raw experimental data are publicly available under the BSD 3-Clause License: https://doi.org/10.6084/m9.figshare.23688879.v1.

ACKNOWLEDGMENTS

This work was funded in part by National Science Foundation grants CCF-2100037 and CNS-2100015.

REFERENCES

[1] Apache Software Foundation. 2023. Apache Ant (version 1.10.13). https://ant.apache.org/.
[2] Apache Software Foundation. 2023. Apache Commons BCEL Issue #367. https://issues.apache.org/jira/browse/BCEL-367.
[3] Apache Software Foundation. 2023. Apache Commons BCEL Issue #368. https://issues.apache.org/jira/browse/BCEL-368.
[4] Apache Software Foundation. 2023. Apache Commons BCEL (version 6.7.0). https://commons.apache.org/proper/commons-bcel/.
[5] Apache Software Foundation. 2023. Apache Maven (version 3.9.2). https://maven.apache.org/.
[6] Apache Software Foundation. 2023. Apache Tomcat (version 10.1.9). https://tomcat.apache.org.
[7] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250. https://doi.org/10.1002/stvr.1486 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1486
[8] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, USA. https://doi.org/10.14722/ndss.2019.23412
[9] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure While Fuzzing. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 1985–2002.
[10] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for

Computing Machinery, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266
[11] Closure Compiler Authors. 2020. Google Closure Issue#3593. https://github.com/google/closure-compiler/issues/3593.
[12] Closure Compiler Authors. 2023. Google Closure Issue#4096. https://github.com/google/closure-compiler/issues/4096.
[13] Closure Compiler Authors. 2023. Google Closure (version v20230502). https://github.com/google/closure-compiler.
[14] Kenneth Alan De Jong. 1975. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* Ph. D. Dissertation. University of Michigan, USA. AAI7609381.
[15] Kenneth A. De Jong and William M. Spears. 1991. An analysis of the interacting roles of population size and crossover in genetic algorithms. In *Parallel Problem Solving from Nature*, Hans-Paul Schwefel and Reinhard Männer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 38–47.
[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, USA.
[17] John H. Holland. 2000. Building Blocks, Cohort Genetic Algorithms, and Hyperplane-Defined Functions. *Evol. Comput.* 8, 4 (dec 2000), 373–391. https://doi.org/10.1162/106365600568220
[18] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Bellevue, WA) *(Security'12)*. USENIX Association, USA, 38.
[19] Paul Holser. 2023. junit-quickcheck. https://github.com/pholser/junit-quickcheck.
[20] Katherine Hough and Jonathan Bell. 2024. Supplemental Materials for "Crossover in Parametric Fuzzing". (1 2024). https://doi.org/10.6084/m9.figshare.24932631.v1
[21] Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 435–445. https://doi.org/10.1145/2568225.2568271
[22] 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* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804
[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* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 438–450. https://doi.org/10.1145/3510003.3510628
[24] 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
[25] LLVM Project. 2023. libFuzzer. https://llvm.org/docs/LibFuzzer.html.
[26] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1949–1966. https://www.usenix.org/conference/usenixsecurity19/presentation/lyu
[27] Björn Mathis, Rahul Gopinath, and Andreas Zeller. 2020. Learning Input Tokens for Effective Fuzzing. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 27–37. https://doi.org/10.1145/3395363.3397348
[28] MDN Contributors. 2018. Mozilla Rhino Issue #397. https://github.com/mozilla/rhino/issues/397.
[29] MDN Contributors. 2018. Mozilla Rhino Issue #405. https://github.com/mozilla/rhino/issues/405.
[30] MDN Contributors. 2018. Mozilla Rhino Issue #406. https://github.com/mozilla/rhino/issues/406.
[31] MDN Contributors. 2018. Mozilla Rhino Issue #409. https://github.com/mozilla/rhino/issues/409.
[32] MDN Contributors. 2023. Mozilla Rhino Issue #1337. https://github.com/mozilla/rhino/issues/1337.
[33] MDN Contributors. 2023. Mozilla Rhino (version 1.7.14). https://github.com/mozilla/rhino.
[34] 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
[35] Michał Zalewski. 2023. American Fuzzing Lop (AFL). https://lcamtuf.coredump.cx/afl/.
[36] Melanie Mitchell, John H. Holland, and S. Forrest. 1992. The royal road for genetic algorithms: Fitness landscapes and GA performance. In *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*. MIT Press, Cambridge, MA, USA, 245–254.
[37] Mountainminds GmbH & Co. KG and Contributors. 2021. JaCoCo Java Code Coverage Library (version 0.8.7). https://github.com/jacoco/jacoco.
[38] Hoang Lam Nguyen and Lars Grunske. 2022. BeDivFuzz: Integrating Behavioral Diversity into Generator-Based Fuzzing. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 249–261. https://doi.org/10.1145/3510003.3510182
[39] Oracle Corporation. 2023. OpenJDK Java Class Library (version 11.0.19). https://openjdk.java.net/.
[40] Oracle Corporation. 2023. Oracle Java Bug Database JDK-8309911. https://bugs.java.com/bugdatabase/view_bug?bug_id=JDK-8309911.
[41] Oracle Corporation. 2023. Oracle Java Bug Database JDK-8309914. https://bugs.java.com/bugdatabase/view_bug?bug_id=JDK-8309914.
[42] Oracle Corporation. 2023. Oracle Java Bug Database JDK-8309915. https://bugs.java.com/bugdatabase/view_bug?bug_id=JDK-8309915.
[43] OSS-Fuzz Contributors. 2023. OSS-Fuzz. https://github.com/google/oss-fuzz.
[44] OW2 Consortium. 2023. ASM (version 9.1). https://asm.ow2.io/.
[45] 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
[46] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576
[47] V. Pham, M. Bohme, A. E. Santosa, A. Caciulescu, and A. Roychoudhury. 2021. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* 47, 09 (sep 2021), 1980–1997. https://doi.org/10.1109/TSE.2019.2941681
[48] Günther R. Raidl and Jens Gottlieb. 2005. Empirical Analysis of Locality, Heritability and Heuristic Bias in Evolutionary Algorithms: A Case Study for the Multidimensional Knapsack Problem. *Evolutionary Computation* 13, 4 (12 2005), 441–475. https://doi.org/10.1162/106365607774666886
[49] 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* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1410–1421. https://doi.org/10.1145/3377811.3380399
[50] Rohan Padhye and JQF Contributors. 2023. jqf-examples (version 2.0). https://central.sonatype.com/artifact/edu.berkeley.cs.jqf/jqf-examples/2.0.
[51] Franz Rothlauf. 2006. *Representations for Genetic and Evolutionary Algorithms*. Springer-Verlag, Berlin, Heidelberg.
[52] 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
[53] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. http://www.jstor.org/stable/1165329
[54] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 724–735. https://doi.org/10.1109/ICSE.2019.00081
[55] Richard A. Watson and Thomas Jansen. 2007. A Building-Block Royal Road Where Crossover is Provably Essential. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation* (London, England) *(GECCO '07)*. Association for Computing Machinery, New York, NY, USA, 1452–1459. https://doi.org/10.1145/1276958.1277224
[56] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. 2019. SLF: Fuzzing without Valid Seed Inputs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 712–723. https://doi.org/10.1109/ICSE.2019.00080
[57] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. 2019. ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery. In *2019 IEEE Symposium on Security and Privacy (SP)*. 769–786. https://doi.org/10.1109/SP.2019.00057