Semantic Fuzzing with Zest

Rohan Padhye
University of California, Berkeley

Caroline Lemieux
University of California, Berkeley

Koushik Sen
University of California, Berkeley

Mike Papadakis
University of Luxembourg

Yves Le Traon
University of Luxembourg

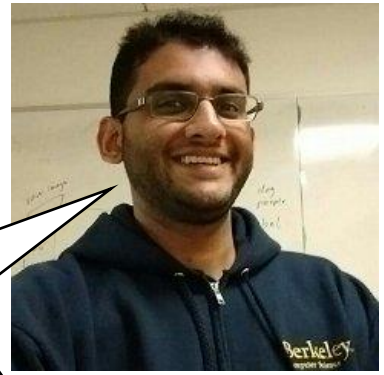
Useless Backstory

Sometime in 2017

I have a grant on finding performance vulnerabilities in Java!

I have students! I built concolic executors for Java! Let's collaborate!

sea of profs

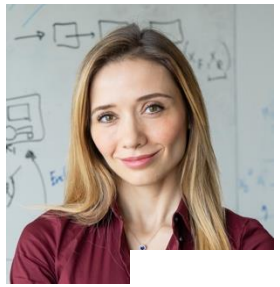


I know Java! I will build the Java fuzzing part of it.

I know AFL! I'll build the performance-fuzzing AFL part!

land of grad students

Later in 2017



That's not really security…. But ok…

Can I do performance fuzzing of C programs as my course project?

Detecting Algorithmic Complexity Attacks via Fuzz Testing

Caroline Lemieux
UC Berkeley

ABSTRACT

Algorithmic complexity attacks are a form of de
tacks which exploit the difference between avera
performance of a deployed algorithm. In this p
an automated attack method which uses fuzz tes
potentially causing complexity attacks. We imple
as an extension of the popular fuzz testing tool .
can be used on C/C++ programs to find time c
or on Java programs to find time and space comp
find our method performs favorably against pre
fuzz testing to find complexity attacks. We also e
of different feedback modes on the performanc
We confirm that our method can automatically fi
Finally, our method automatically found new co
bilities in JDK methods, awaiting further confirm

1 INTRODUCTION

Many popular algorithms suffer from highly vari
between their best, average, and worst case in
examples include sorting algorithms. Insertion
runtime in $O(n)$ and worst-case in $O(n^2)$; quick
case runtime in $O(n \log n)$ and worst-case in $O($
of algorithms with such variable performance ca
rity vulnerabilities in a system which takes in
performance bottlenecks are major, the system
ble to major slowdown or even a DoS-style attac
needs to do is figure out which inputs trigger
the program. Such attacks are known as *algori*
attacks. Figure 1 outlines the attack procedure. U

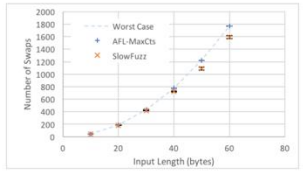


Figure 4: Maximum number of swaps performed while sorting generated inputs with insertion sort for SlowFuzz and AFL-MaxCts. Points illustrate average over 5 experiments.

Figure 5: Maximum number of branches executed while sorting generated inputs for SlowFuzz and AFL-MaxCts. Points illustrate average over 5 experiments.

all programs, and should be closely correlated to total instruction count. We see the results for insertion sort in Figure 5. We see

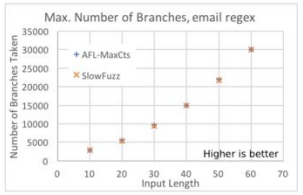


Figure 6: Maximum number of branches executed matching generated inputs to an email regex for SlowFuzz and AFL-MaxCts. Points illustrate average over 5 experiments.

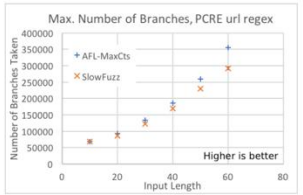


Figure 7: Maximum number of branches executed matching generated inputs to a URL regex for SlowFuzz and AFL-MaxCts. Points illustrate average over 5 experiments.

AFL-MaxCts to find inputs exercising quadratic match time (typi-

Early (Early) 2018

Wow! Distinguished paper award! I don't need even to help you guys write papers anymore!

PERFFUZZ: Automatically Generating Pathological Inputs

Anonymous Author(s)

ABSTRACT

Performance problem
programs are provided
A large body of work
via statistical profiling
inputs in the first pla
tomatically generate
across program locati
Fuzz generates inputs
Unlike previous approa
characteristic such as t
multi-dimensional fee
tion counts for all pro
find a variety of inputs
and (2) generate inpu
than previous approac
also effective at gener
complexity vulnerabili
a popular coverage-gu
four real-world C prog
We find that PerfFuzz
hat exercise the most

do these inputs come from? The most commonly chosen sources include (1) specially hand-crafted performance tests [44, 46], (2)

[ISSTA 2018] Accepted paper #123 "PerfFuzz: Automatically Generating..." External

ISSTA 2018 HotCRP <noreply@issta18.hotcrp.com> Sun, Apr 29, 2018, 11:54 PM
to Caroline, Dawn, Koushik, Rohan, eric.bodden

Dear authors,

The 27th Interr
2018) program
been accepted

Title: PerfF
Authors: Ca
Rohar
Koush
wn
tion

We can submit this performance fuzzing stuff just for C to ISSTA!

[ISSTA 2018] Distinguished Paper Award for paper #123 "PerfFuzz: Automatically Generating..." External

ISSTA 2018 HotCRP <noreply@issta18.hotcrp.com> Wed, May 30, 2018, 10:42 PM
to Caroline, Dawn, Koushik, Rohan, eric.bodden, f.tip, karim.ali

Dear authors,

I am very pleased to inform you that your paper entitled...

PerfFuzz: Automatically Generating Pathological Inputs

... has been awarded an ACM SIGSOFT Distinguished Paper Award! Congratulations!

Any ACM conference can award up to 10% of accepted papers that way. At ISSTA 2018, three papers will be receiving such an award.

Later in 2018

So…. What about the Java thing?

JQF: Generating Semantically Valid Test Inputs using Parametric Generators and Coverage-Guided Fuzzing

Anonymous Author(s)

ABSTRACT

Programs expecting str

ESEC/FSE 2018 - Notification for paper 217 - REJECT

My major concern is that it is unclear what exactly this paper has achieved. In the main technical part, Section 3.1 describes mutating a byte sequence. Obviously, any byte sequence could be converted into a text (or a program) deterministically depending on the mapping. The key question here is why one wants to mutate a byte sequence rather than the text in the first place.

18, 10:13 AM

Maven, Ant, the Google
attains statistically sign
valid inputs compared t
of these 4 benchmarks,

ACM Reference Format:
Anonymous Author(s). 201
using Parametric Generato
of The 26th ACM Joint Eur
posium on the Foundations
United States, 4–9 Novembe
https://doi.org/10.1145/nn

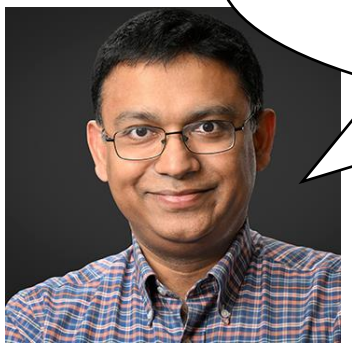
age-Guided Fuzzing

(1) What exactly has been achieved? In particular, what can JQF do while the previous approaches (e.g., the generator described in Figure 3) cannot?

The competition w
submissions (i.e.,
process, with eacl
Sweden and discu
conditionally acce

We enclose below the reviews on your paper, which also include the META-REVIEW which summarizes the discussion on your paper.

Later (Later) in 2018



Zest: Validity Fuzzing and Parametric Generators for Effective Random Testing

OK
...
Zest!

Abstract—Programs
of both a *syntactic an*
into an internal data
which conducts check
core logic of the progr
like coverage-guided f
tend to produce inpu
two stages. We propos
effectively explores the
Zest combines two k
introduce *validity fuzz*
semantically valid inpu
ators, which convert in
as a sequence of numb
syntactically valid XM
mutations to map to
test inputs. We imple
AFL and QuickCheck,
tools, on six real-worl
BCEL, ScalaChess, t
Rhino. We find that
semantic analysis sta
we find 18 new bugs
that are uniquely four

ICSE 2019 notification for paper 398

Technical Track <icse2019-technical@easychair.org> Tue, Dec 11, 2018, 10:11 PM

to Caroline

Dear Caroline,

Thank you for submitting your paper

Zest: Validity Fuzzing and Parametric Generators for Effective Random Testing

to the ICSE 2019 Technical Track.

We regret to inform you that your paper has not been accepted to the program. Each paper was reviewed by at least 3 Program Committee (PC) members and the decision overseen by a member of the Program Board (PB). Out of 529 submissions, 109 papers were accepted (an acceptance rate of 21%). The reviews as well as a meta-review written by the PB member summarizing the discussion are included below. We hope that you find the feedback provided in the reviews and meta-review useful, and that it will help you to revise your work for a future submission.

Again, Confusion about “Parametric Generators”

Ben, can you read this section on parametric generators? And tell me how confusing it is on a scale of 1-10

Ok.

It is 7/10 confusing

ICSE'19 Version

B. Parametric Generators

We illustrate the intuition behind parametric generators by returning to the XML generator from Figure 2. Fundamentally, the behavior of the generator depends on the values produced by the pseudo-random number source that it is given, referenced by variable `random` in the example.

Let us consider one particular instance where `random` produces the sequence σ_1 of pseudo-random numbers, with values: $3, 102, 111, 111, 2, 3, \ldots, 0, 0$. We can see how the numbers returned by `random`—i.e. those in σ_1—influence the XML generator's behavior by looking at the generator's execution trace, here simplified to a sequence of line numbers from Figure 2 and the effect on the generated XML:

(Line 27) Root node: name length = **3**
(Line 30) Root node: name[0] = **102** (ASCII **'f'**)
(Line 30) Root node: name[1] = **111** (ASCII **'o'**)
(Line 30) Root node: name[2] = **111** (ASCII **'o'**)
(Line 14) Root node: number of children = **2**
  (Line 11) First child: name length = **3**
  ⋮
  (Line 19) Second child: embed text = **0** (False)
(Line 19) Root node: embed text = **0** (False)

And the XML produced by this instance, say x_1, looks like:

```
<foo><bar>Hello</bar><baz /></foo>
```

Notice that the generated test-input is simply a function of the numbers produced by the pseudo-random source. A *parametric generator* is a function that, instead of relying on parameters from a random number generator, takes a sequence of numeric values such as σ_1—the *parameter sequence*—and produces a structured input, such as the XML x_1.

The following *key observation* allows us to use parametric generators to map low-level mutations in the parameter space to high-level mutations in the structured input space. If a parameter sequence σ, which leads to the generation of input x, is slightly mutated to produce a new sequence σ', then the corresponding generated input x' will be a structured mutant of x in the space of synctactically valid inputs. That is, if σ' is similar to σ, then x' will likely share some structure with x. Therefore, by mutating the stream of parameters fed to a parametric generator, we can perform high-level structured mutations on inputs while retaining their syntactic validity.

To illustrate this, suppose that the second value in the sequence σ_1 above is randomly set to 87, producing the sequence σ_2: 3, **87**, 111, 111, 2, 3, …, 0, 0. When σ_2 is passed to the parametric generator, the generator produces x_2:

```
<Woo><bar>Hello</bar><baz /></Woo>
```

Mutational fuzzing with parametric generators: Concretely, we combine parametric generators and validity fuzzing in the following way. Let $p_A : A \to T$ be a program that takes input of type A and produces a result of type T. In the example from Figure 3, the test harness accepts inputs of type `String` and produces a test result. Therefore, A is the set of all strings, and $T = \{\text{pass}, \text{fail}, \text{invalid}\}$. Let $g_A : \Sigma \to A$ represent a parametric generator that takes a parameter sequence $\sigma \in \Sigma$ and produces a value in A. The generator in Figure 2 can be represented as a parametric generator where A is the set of all strings. Now, we can compose g_A and p_A to produce a new program $p_\Sigma : \Sigma \to T$ that takes as input a parameter sequence and produces a test result: $p_\Sigma = p_A \circ g_A$.

We can now run Algorithm 2 with the program $p := p_\Sigma$, and $I := \{\sigma_r\}$, an initial parameter sequence σ_r that is randomly generated. Thus, the fuzzing algorithm mutates and saves *parameter sequences* instead of test inputs, while using feedback from the execution of the underlying program and the validity of the inputs produced by the parametric generators. At the end of the fuzzing loop, the returned corpus $\mathcal{V}$ now contains parameter sequences corresponding to valid inputs. Those inputs can be retrieved as $\mathcal{V}_A = \{g_A(\sigma) \mid \sigma \in \mathcal{V}\}$. In our experimental evaluation, we refer to this combination of parametric generators and validity fuzzing as Zest_{VG}.

ISSTA'19 Version

3.1 Parametric Generators

Before defining parametric generators, let us return to the random XML generator from Figure 2. Let us consider a particular path through this generator, concentrating on the calls to `nextInt`, `nextBool`, and `nextChar`. The following sequence of calls will be our running example (some calls ommitted for space):

Call → result	Context
`random.nextInt(1, MAX_STRLEN)` → 3	Root: name length (Line 26)
`random.nextChar()` → 'f'	Root: name[0] (Line 29)
`random.nextChar()` → 'o'	Root: name[1] (Line 29)
`random.nextChar()` → 'o'	Root: name[2] (Line 29)
`random.nextInt(MAX_CHILDREN)` → 2	Root: # children (Line 13)
`random.nextInt(1, MAX_STRLEN)` → 3	Child 1: name length (Line 26)
⋮	
`random.nextBool()` → False	Child 2: embed text? (Line 19)
`random.nextBool()` → False	Root: embed text? (Line 19)

The XML document produced when the generator makes this sequence of calls looks like:

x_1 = `<foo><bar>Hello</bar><baz /></foo>`.

In order to produce random typed values, the implementations of `random.nextInt`, `random.nextChar`, and `random.nextBool` rely on a pseudo-random source of *untyped* bits. We call these untyped bits "*parameters*". The parameter sequence for the example above, annotated with the calls which consume the parameters, is:

σ_1 = `0000 0010` `0110 0110` ... `0000 0000` .

`nextInt(1,...)→3` `nextChar()→'f'` `nextBool()→False`

For example, here the function `random.nextInt(`a, b`)` consumes eight bit parameters as a byte, n, and returns $n \,\%\, (b - a) + a$ as a typed integer. For simplicity of presentation, we show each `random.nextXYZ` function consuming the same number of parameters, but they can consume different numbers of parameters.

We can now define a ***parametric generator***. A parametric generator is a function that takes a sequence of untyped parameters such as σ_1—the *parameter sequence*—and produces a structured input, such as the XML x_1. A parametric generator can be implemented

(2) *Bit-level mutations on untyped parameter sequences correspond to high-level structural mutations in the space of syntactically valid inputs.*

Observation (1) is true by construction. The `random.nextXYZ` functions are implemented to produce correctly-typed values no matter what bits the pseudo-random source–or in our case, the parameters—provide. Every sequence of untyped parameter bits correspond to some execution path through the generator, and therefore every parameter sequence maps to a syntactically valid input. We describe how we handle parameter sequences that are longer or shorter than expected with the example sequences σ_3 and σ_4, respectively, below.

To illustrate observation (2), consider the following parameter sequence, σ_2, produced by mutating just a few bits of σ_1:

σ_2 = `0000 0010` `0101 0111` `... 0000 0000`.

`nextChar()→'W'`

As indicated by the annotation, all this parameter-sequence mutation does is change the value returned by the second call to `random.nextChar()` in our running example from `'f'` to `'W'`. So the generator produces the following test-input:

x_2 = `<Woo><bar>Hello</bar><baz /></Woo>`.

Now in 2019

Semantic Fuzzing with Zest

Anonymous Author(s)

ABSTRACT

Programs expecting structured inputs
tic analysis stage, which parses raw i
stage, which conducts checks on th
the core logic of the program. Gener
lineage of QuickCheck are a promis
syntactically valid test inputs for the
fectively explore the semantic analy
tedious manual tuning of the genera
nique which automatically guides Q
generators to better explore the sema
grams. Zest converts random-input
parametric generators. We present th
in the untyped parameter domain m
the input domain. Zest leverages this
based mutational fuzzing, directed by
of code coverage and input validity
and evaluate it against AFL and Q
benchmarks: Maven, Ant, BCEL, the
Mozilla Rhino. We find that Zest out
in terms of code coverage within th
these benchmarks. Further, we find
analysis stage across these benchma
technique in finding these bugs, req
average to find each such bug.

[ISSTA 2019] Paper #272 "Semantic Fuzzing with Zest" External

ISSTA 2019 HotCRP <noreply@issta19.hotcrp.com> Wed, May 1, 2019, 3:11AM

to Caroline, Koushik, Mike, Rohan, Yve, amoeller

Dear authors,

The ISSTA 2019 program committee is delighted to inform you that your paper #272 has been accepted to appear in the conference.

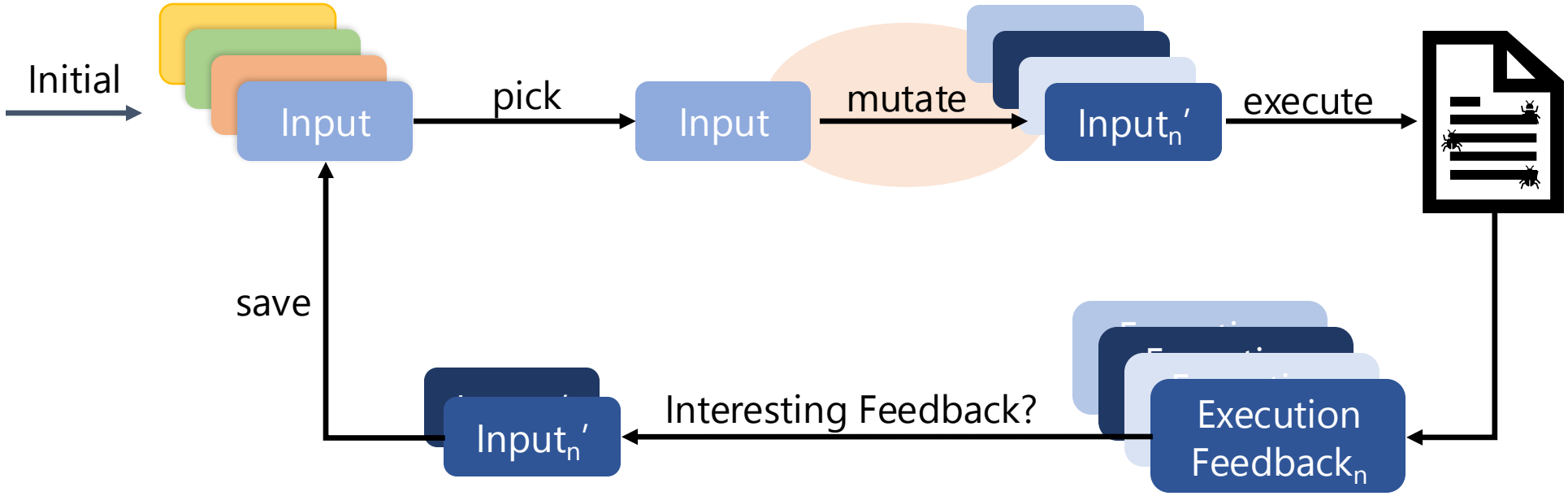
Title: Semantic Fuzzing with Zest
Authors: Rohan Padhye (University of California, Berkeley)
Caroline Lemieux (University of California, Berkeley)
Koushik Sen (University of California, Berkeley)
Mike Papadakis (University of Luxembourg)
Yves Le Traon (University of Luxembourg)
Paper site: https://issta19.hotcrp.com/paper/272

Your paper was one of 29 accepted out of 142 submissions. (3 more submissions have been conditionally accepted.) Congratulations!

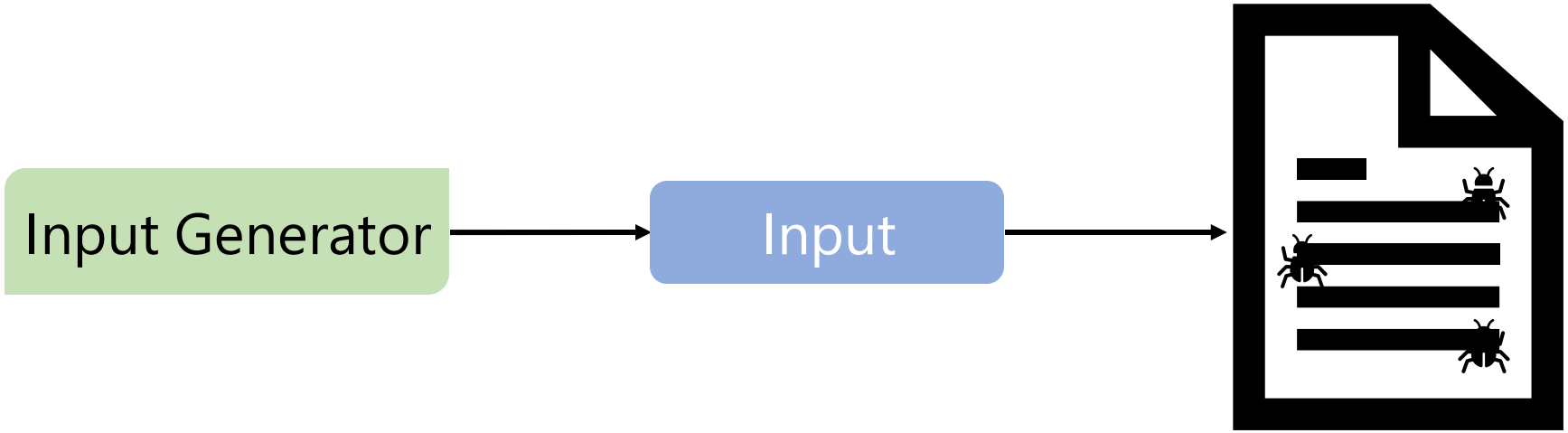
Technique

Can we get higher-level mutations?

with more information about input structure?



Generators as Input Structure Specification

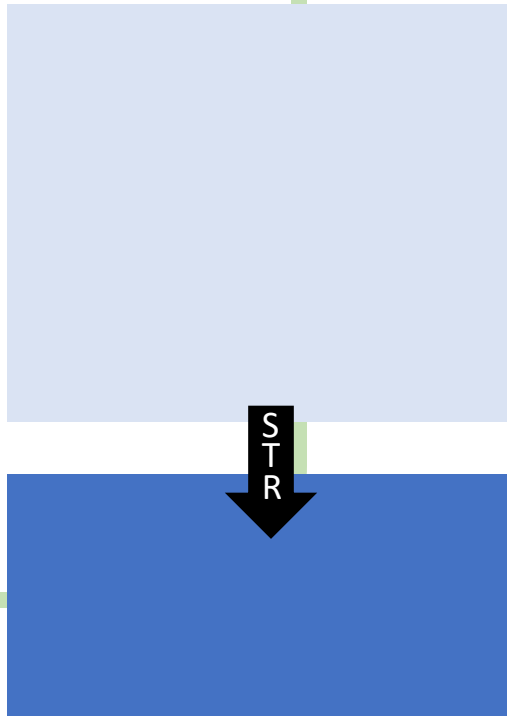


How to Get Mutations?

```
def genXML(random):
    tag = random.choice(tags)
    node = XMLElement(tag)
    num_child = random.nextInt(0, MAX_CHILDREN)
    for i in range(0, num_child):
        node.addChild(genXML(random))
    if random.nextBoolean():
        node.addText(random.nextString())
    return node
```

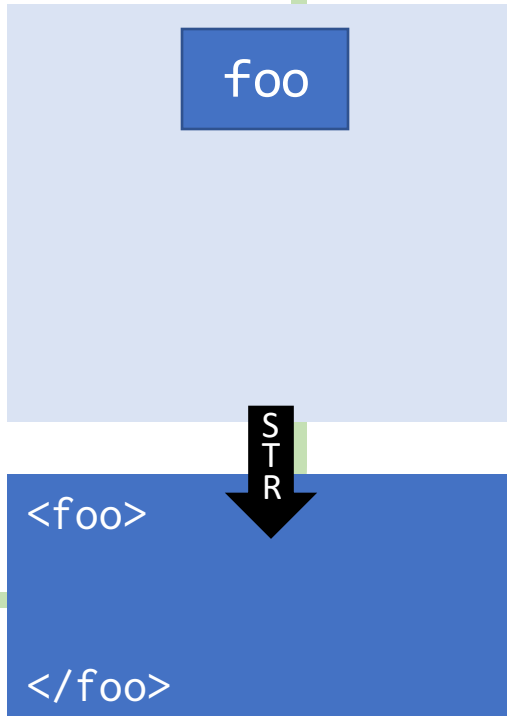
Generator: Source of Randomness → Input

```
def genXML(random):
    tag = random.choice(tags)
    node = XMLElement(tag)
    num_child = random.nextInt(0, MAX_CHILDREN)
    for i in range(0, num_child):
        node.addChild(genXML(random))
    if random.nextBoolean():
        node.addText(random.nextString())
    return node
```



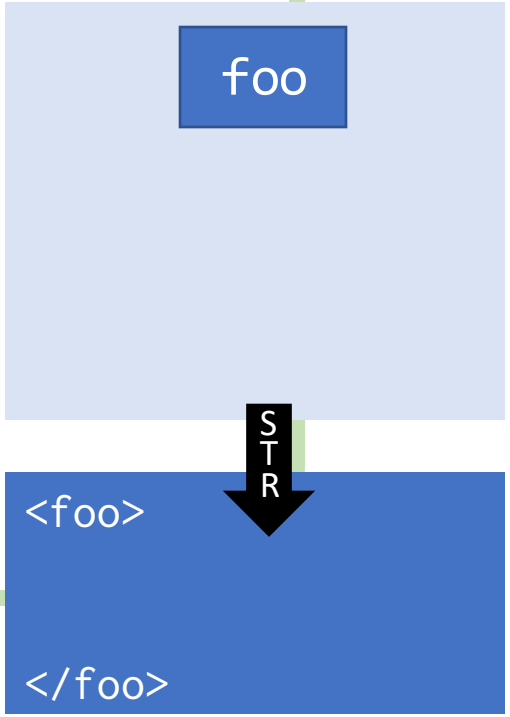
Generator: Source of Randomness → Input

```
def genXML(random):
    tag = random.choice(tags)
    node = XMLElement(tag)
    num_child = random.nextInt(0, MAX_CHILDREN)
    for i in range(0, num_child):
        node.addChild(genXML(random))
    if random.nextBoolean():
        node.addText(random.nextString())
    return node
```



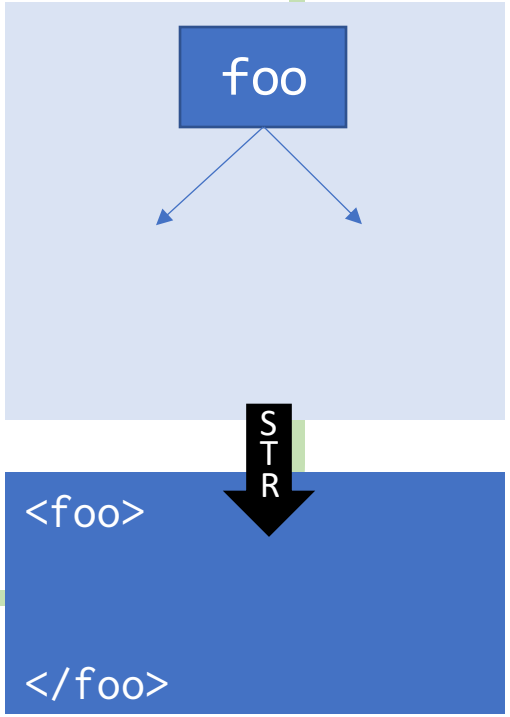
Generator: Source of Randomness → Input

```
def genXML(random):
    tag = random.choice(tags)
    node = XMLElement(tag)
    num_child = random.nextInt(0, MAX_CHILDREN)
    for i in range(0, num_child):
        node.addChild(genXML(random))
    if random.nextBoolean():
        node.addText(random.nextString())
    return node
```



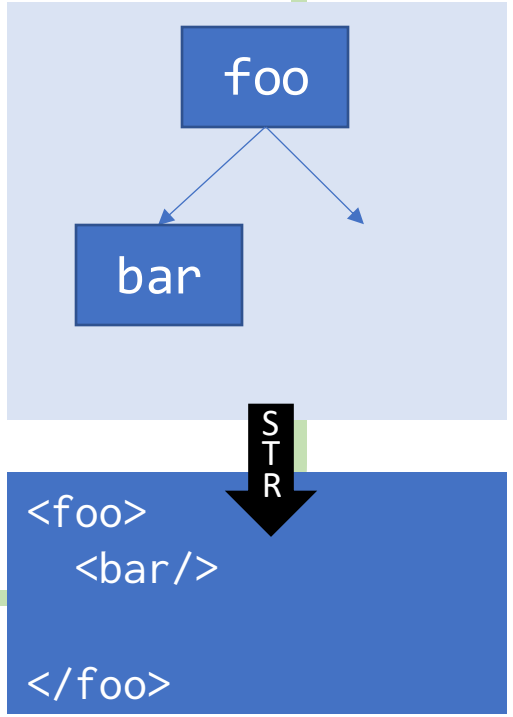
Generator: Source of Randomness → Input

```
def genXML(random):
    tag = random.choice(tags)
    node = XMLElement(tag)
    num_child = random.nextInt(0, MAX_CHILDREN)
    for i in range(0, num_child):
        node.addChild(genXML(random))
    if random.nextBoolean():
        node.addText(random.nextString())
    return node
```



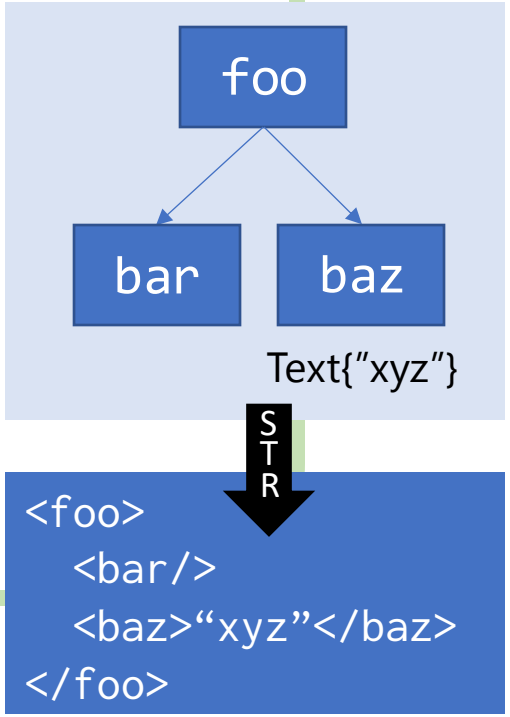
Generator: Source of Randomness → Input

```
def genXML(random):
    tag = random.choice(tags)
    node = XMLElement(tag)
    num_child = random.nextInt(0, MAX_CHILDREN)
    for i in range(0, num_child):
        node.addChild(genXML(random))
    if random.nextBoolean():
        node.addText(random.nextString())
    return node
```



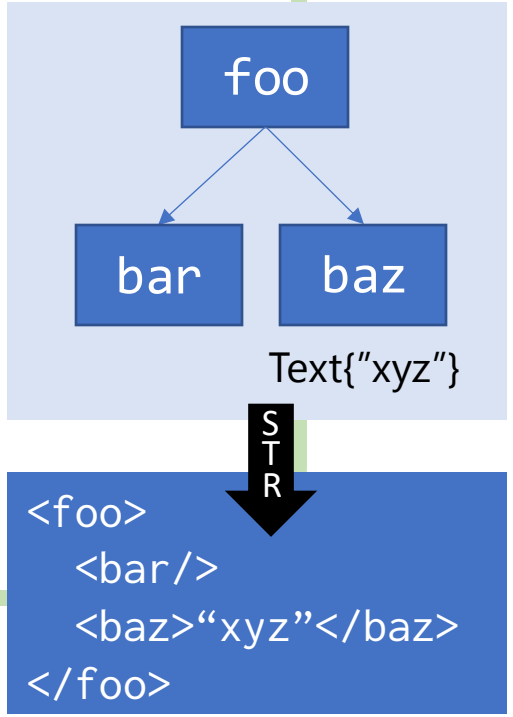
Generator: Source of Randomness → Input

```
def genXML(random):
    tag = random.choice(tags)
    node = XMLElement(tag)
    num_child = random.nextInt(0, MAX_CHILDREN)
    for i in range(0, num_child):
        node.addChild(genXML(random))
    if random.nextBoolean():
        node.addText(random.nextString())
    return node
```



Generator: Source of Randomness → Input

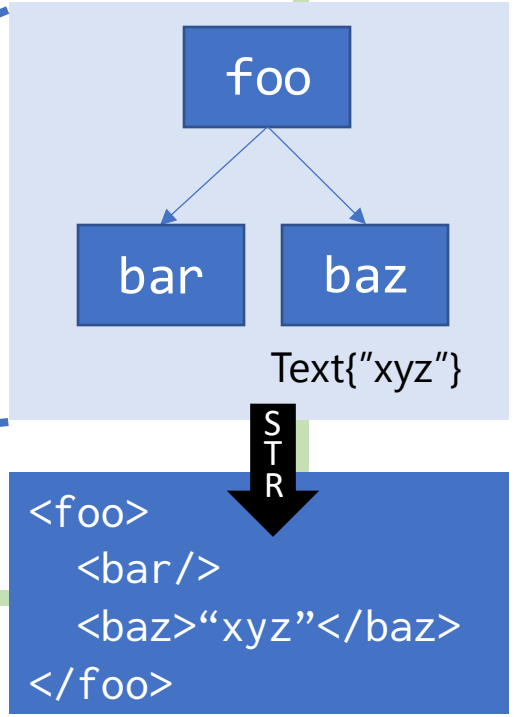
```
def genXML(random):
    tag = random.choice(tags)
    node = XMLElement(tag)
    num_child = random.nextInt(0, MAX_CHILDREN)
    for i in range(0, num_child):
        node.addChild(genXML(random))
    if random.nextBoolean():
        node.addText(random.nextString())
    return node
```



Source of Randomness == Infinite Bit-Sequence

pseudo-random bits: 0000 0011 0110 0110 0110 1111 0110 1111 0000 0010 …

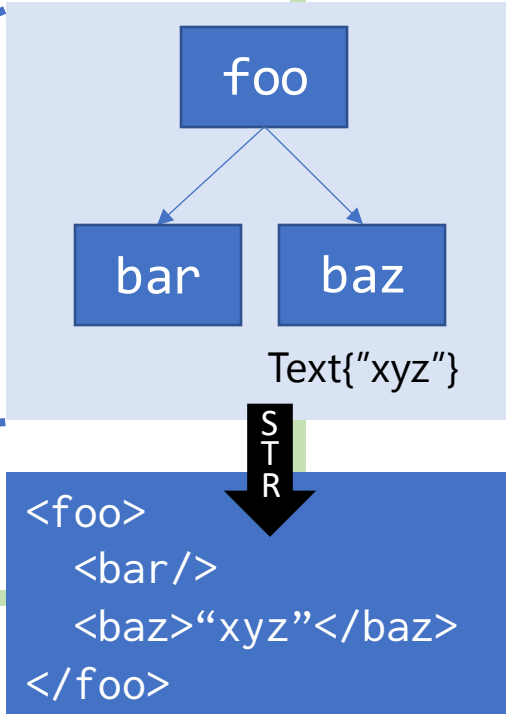
```
def genXML(random):
    tag = random.choice(tags)
    node = XMLElement(tag)
    num_child = random.nextInt(0, MAX_CHILDREN)
    for i in range(0, num_child):
        node.addChild(genXML(random)
    if random.nextBoolean():
        node.addText(random.nextString())
    return node
```



Bit Mutations → Structured Input Mutations

pseudo-random bits: 0000 0011 0110 0110 0110 1111 0110 1111 0000 0010 …

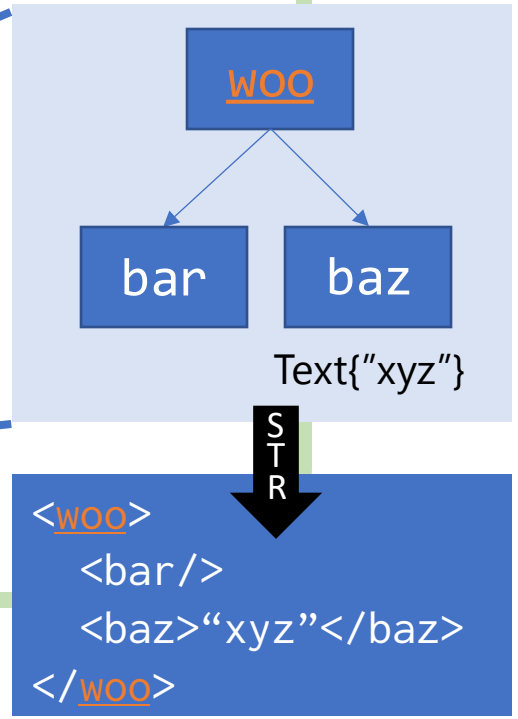
```
def genXML(random):
    tag = random.choice(tags)
    node = XMLElement(tag)
    num_child = random.nextInt(0, MAX_CHILDREN)
    for i in range(0, num_child):
        node.addChild(genXML(random)
    if random.nextBoolean():
        node.addText(random.nextString())
    return node
```



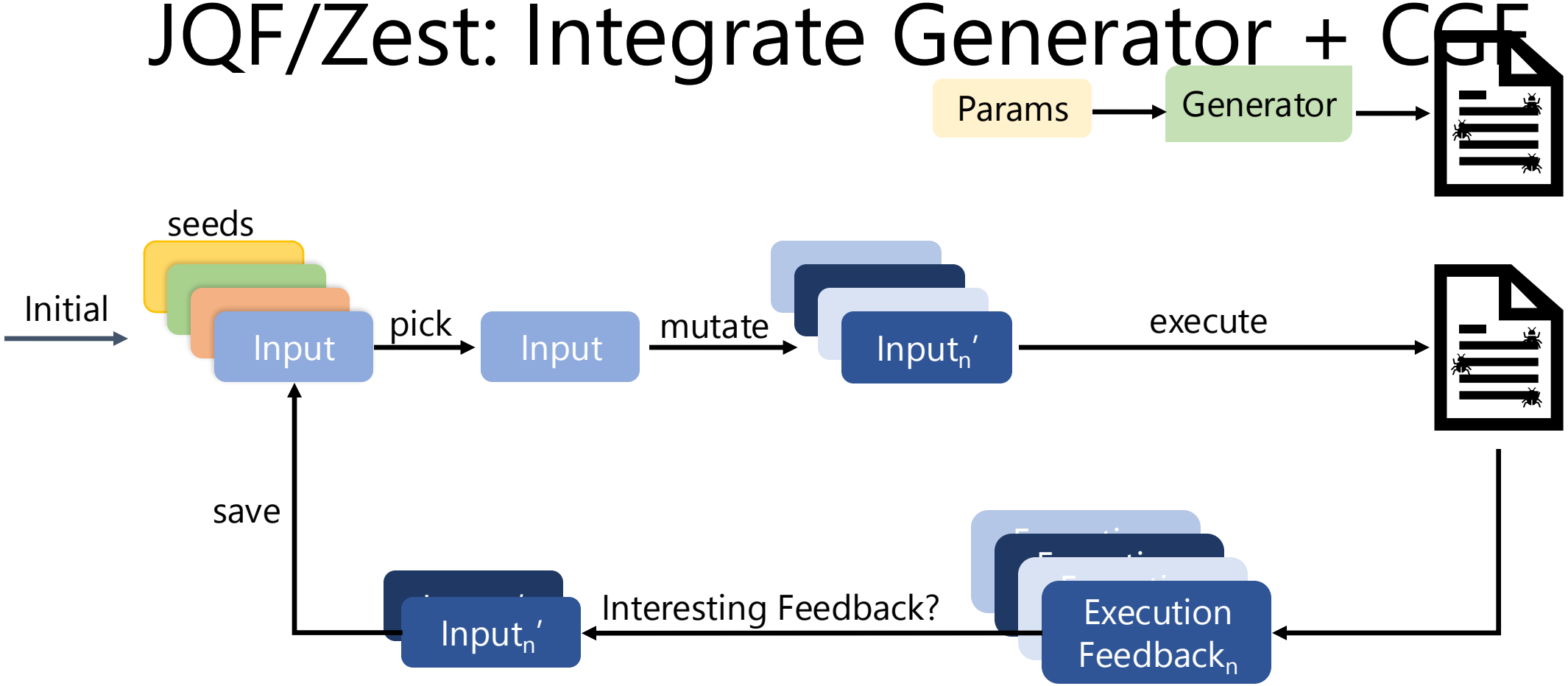
Bit Mutations → Structured Input Mutations

pseudo-random bits: 0000 0011 0101 0111 0110 1111 0110 1111 0000 0010 …

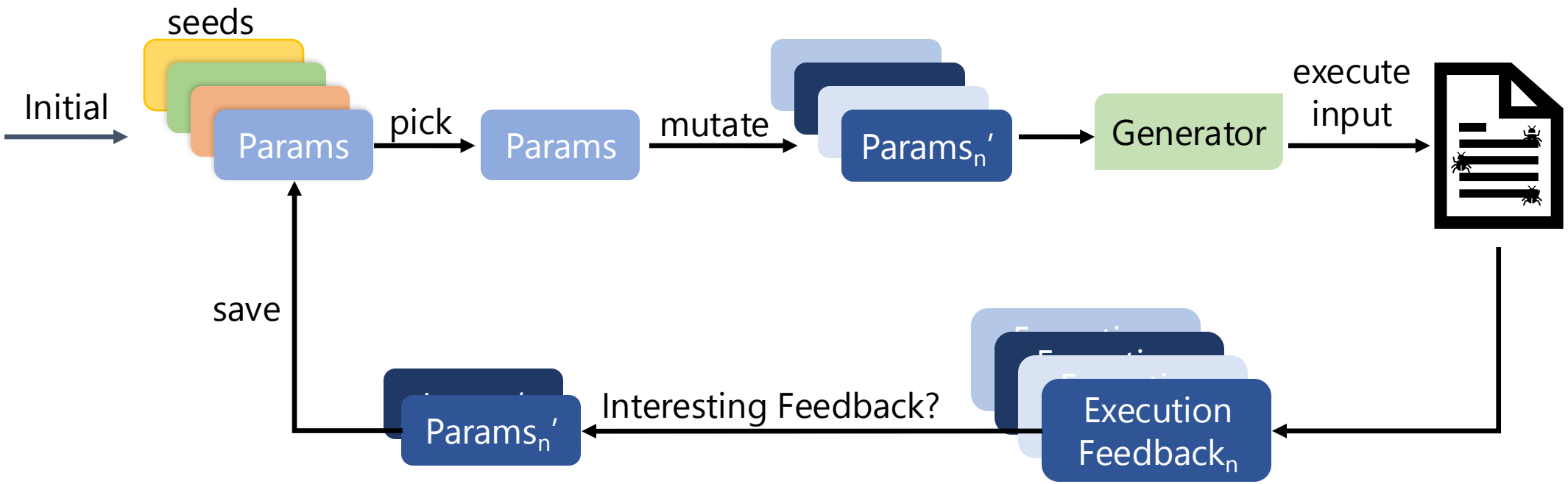
```
def genXML(random):
    tag = random.choice(tags)
    node = XMLElement(tag)
    num_child = random.nextInt(0, MAX_CHILDREN)
    for i in range(0, num_child):
        node.addChild(genXML(random))
    if random.nextBoolean():
        node.addText(random.nextString())
    return node
```



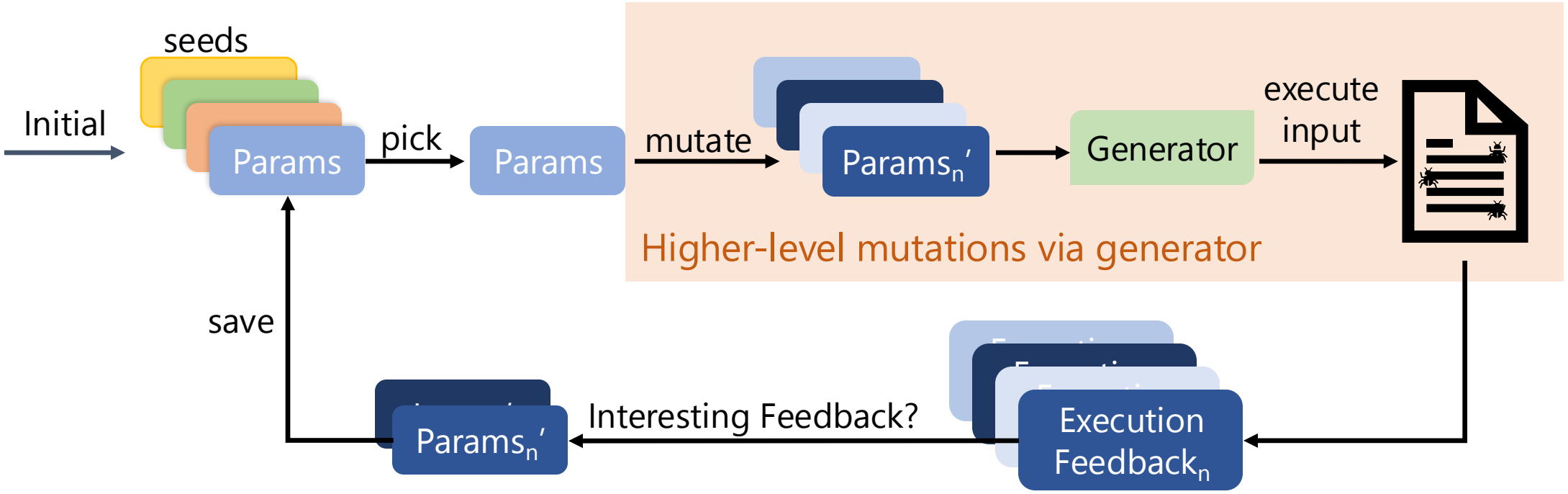
JQF/Zest: Integrate Generator + CGF



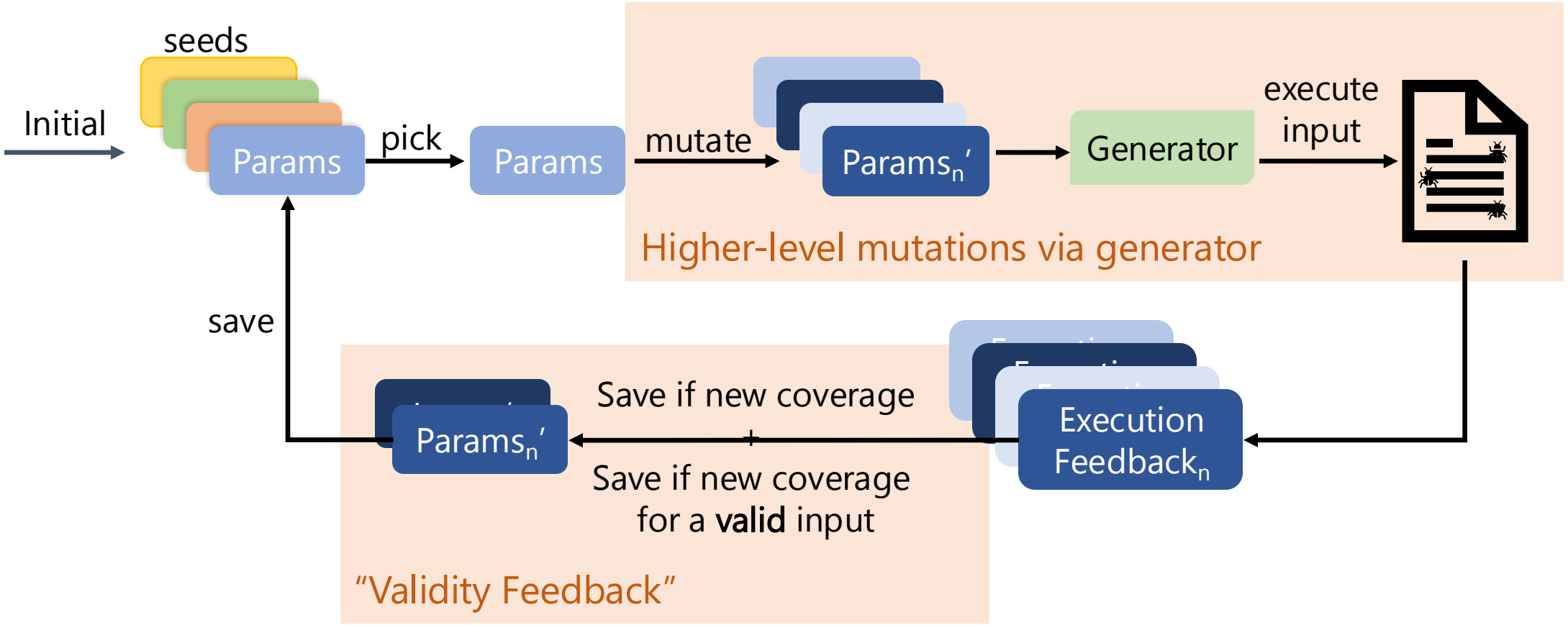
JQF/Zest: Integrate Generator + CGF



JQF/Zest: Integrate Generator + CGF



Zest: Add Validity Feedback Based on `assume`



What's valid? assume?

P in P(x) => Q(x)

```
1  @Property
2  void testProgram(@From(XMLGenerator.class) XMLDocument xml) {
3    Model model = readModel(xml.toString());
4    assume(model != null); // validity
5    assert(runModel(model) == success);
6  }
```

P(xml):
readModel(xml.ToString()) != null
Q(xml):
runModel(readModel(xml.ToString)) == success

Figure 3: A junit-quickcheck property that tests an XML-based component.

Use of S in Mutate()

Algorithm 1 Coverage-guided fuzzing.

Input: program p, set of initial inputs I
Output: a set of test inputs and failing inputs

```
1:  S ← I
2:  F ← ∅
3:  totalCoverage ← ∅
4:  repeat
5:    for input in S do
6:      for 1 ≤ i ≤ NUMCANDIDATES(input) do
7:        candidate ← MUTATE(input, S)
8:        coverage, result ← RUN(p, candidate)
9:        if result = FAILURE then
10:         F ← F ∪ candidate
11:       else if coverage ⊈ totalCoverage then
12:         S ← S ∪ {candidate}
13:         totalCoverage ← totalCoverage ∪ coverage
14: until given time budget expires
15: return S, F
```

The number of new inputs to generate in this round (Line 6) is determined by an implementation-specific heuristic. CGF generates new inputs by applying one or more random mutation operations on the base input (Line 7). These mutations may include operations that combine subsets of other inputs in S. The given program is

Are bit mutations smart at all?

- No.
- We had a "smarter" mutat
  big part of the technical c
- But it was no better than t

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

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

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

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

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

Abstract

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

type or input-format structure. Parametric generators [3, 11, 21, 23, 29–31, 36] enable mutations to be performed on inputs produced by such generators. This unlocks the benefits of coverage-guided grey-box fuzzing [1, 7, 16, 17], which incorporate a feedback loop to guide input generation.

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

By providing the byte-sequence decoded by the generator to a conventional mutation-based fuzzing algorithm, parametric generators get structured mutations "for free". Fig. 2b shows how a

Discussion

First Thoughts/Opinions

- What did you like most/least about the paper?

How Were Benchmarks Chosen?

- Disclaimer: I don't entirely remember
- This was the first "Java fuzzing" work
- XML generator we had early on
- We had some other benchmarks, but Zest was not outperforming quickcheck on those ("too simple")
- Koushik told Rohan: "write a javascript generator today and show me if it works"

Abandoned Benchmarks?

- Generators: https://github.com/rohanpadhye/JQF/tree/master/examples/src/main/java/edu/berkeley/cs/jqf/examples
- Fuzz drivers/properties: https://github.com/rohanpadhye/JQF/tree/master/examples/src/test/java/edu/berkeley/cs/jqf/examples
- Our experience confirms: When a generator covers all the space of inputs (e.g., graph examples), full coverage is easy to get

How Were Benchmarks Chosen?

- Disclaimer: I don't entirely remember
- This was the first "Java fuzzing" work
- XML generator we had early on
- We had some other benchmarks, but Zest was not outperforming quickcheck on those ("too simple")

- Koushik told Rohan: "write a javascript generator today and show me if it works"

Once a Benchmark is Established…

RLCheck (us, ICSE'20)

Benchmarks. We compare the techniques on four real-world Java benchmarks used in the original evaluation of Zest [38]: Apache Ant, Apache Maven, Google Closure Compiler, and Mozilla Rhino. These benchmarks rely on two generators: Ant and Maven use an XML generator, whereas Closure and Rhino use a generator for each of these four bench-

BeDivFuzz (not us, ICSE'22)

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

Confetti (

metric fuzzer JQF-Zest [66] and use the same suite of benchmark programs, given that we built CONFETTI on top of JQF-Zest. Where possible, we used the latest version of the target software that still contained the bugs detected by JQF-Zest in the original work. Following best practices, we study both CONFETTI's ability to explore program branches (*e.g.*, coverage) in comparison to JQF-Zest, and its ability to find new and previously-known bugs [50].

Zeugma (not us, ICSE'24)

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 eva

these subje

subject available in

tions were made to

and Nguyen and G

newer subject versic

included with JQF (

classes. We change

increase the maxim

Kukucka et al. [23]

cise certain functior

generator itself.

Mu2 (RP's group, ISSTA'23)

of each

Benchmarks. We consider five real-world Java programs:[2]

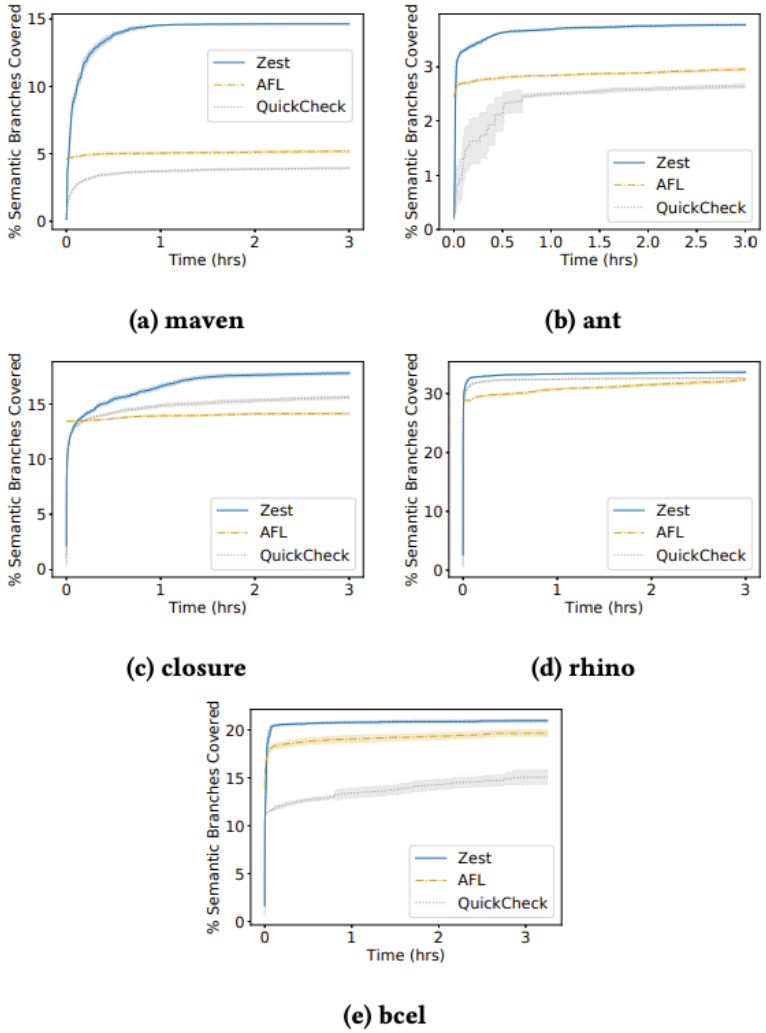
(1) ChocoPy [7, 61] reference compiler (~6K LoC): The test driver (reused from [75]) reads in a program in ChocoPy (a statically typed dialect of Python) and runs the semantic analysis stage of the ChocoPy reference compiler to return a type-checked AST object.
(2) Gson [29] JSON Parser (~26K LoC): The test driver parses a input JSON string and returns a Java object output.
(3) Jackson [22] JSON Parser (~49K LoC): The test driver acts similar to that of Gson.
(4) Apache Tomcat [3] WebXML Parser (~10K LoC): The test driver parses a string input and returns the WebXML representation of the parsed output.
(5) Google Closure Compiler [30] (~250K LoC): The test driver (reused from [59] and [75]) takes in a JavaScript program and performs source-to-source optimizations. It then returns the optimized JavaScript code.

Syntactic Bugs

- https://drive.google.com/file/d/1GCODL1Y4_DagerQ0mq2CbtpTzOljPo97/view

A Note on Performance

- When we run QuickCheck for 3 hours, should we run it with coverage feedback?
- We did in Zest
- Is this a reasonable decision? Why/why not?



Bug ID	Exception	Tool	Mean Time to Find (shorter is better)	Reliability
ant (B)	IllegalStateException	**Zest**	(99.45 sec)	100%
		AFL	(6369.5 sec)	10%
		QC	(1208.0 sec)	10%
closure (C)	NullPointerException	**Zest**	(8.8 sec)	100%
		AFL	(5496.25 sec)	20%
		QC	(8.8 sec)	100%
closure (D)	RuntimeException	**Zest**	(460.42 sec)	60%
		AFL	✗	0%
		QC	✗	0%
closure (U)	IllegalStateException	**Zest**	(534.0 sec)	5%
		AFL	✗	0%
		QC	✗	0%
rhino (G)	IllegalStateException	**Zest**	(8.25 sec)	100%
		AFL	(5343.0 sec)	20%
		QC	(9.65 sec)	100%
rhino (F)	NullPointerException	Zest	(18.6 sec)	100%
		AFL	✗	0%
		QC	(9.85 sec)	100%
rhino (H)	ClassCastException	**Zest**	(245.18 sec)	85%
		AFL	✗	0%
		QC	(362.43 sec)	35%
rhino (J)	VerifyError	**Zest**	(94.75 sec)	100%
		AFL	✗	0%
		QC	(229.5 sec)	80%
bcel (O)	ClassFormatException	Zest	(19.5 sec)	100%
		AFL	(5.85 sec)	100%
		QC	(142.1 sec)	100%
bcel (N)	AssertionViolatedException	**Zest**	(19.32 sec)	95%
		AFL	(1082.22 sec)	90%
		QC	(15.0 sec)	5%

- 10 new bugs:
  - Zest wins on 6
  - QC wins on 1
  - Zest + QC Ties on 2

RLCheck (ICSE'20)

- Goal was to have a blackbox approach to validity fuzzing
  - Pitch was: good to generate as many different valid input as possible
- Ran RLCheck + QuickCheck without coverage feedback for 5 mins, replayed same # of inputs generated

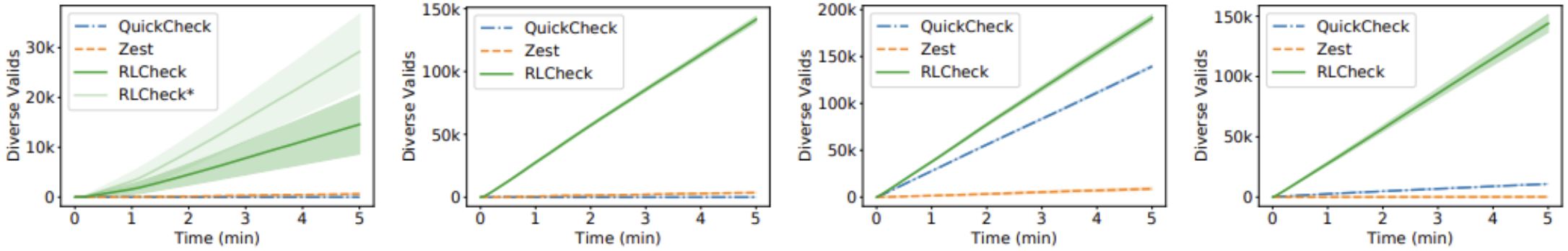


Figure 7: Number of diverse valid inputs (i.e. inputs with different traces) generated by each technique. Higher is better.

Bug discovery comp
(RLCheck: 5 min timeout)

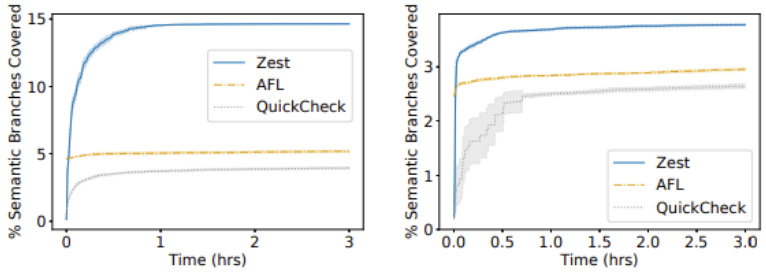
Bug ID	Exception	Tool	Mean Time to Find (shorter is better)	Reliability
ant (B)	IllegalStateException	Zest	(99.45 sec)	100%
		AFL	(6369.5 sec)	10%
		QC	(1208.0 sec)	10%
closure (C)	NullPointerException	Zest	(8.8 sec)	100%
		AFL	(5496.25 sec)	20%
		QC	(8.8 sec)	100%
rhino (G)	IllegalStateException	Zest	(8.25 sec)	100%
		AFL	(5343.0 sec)	20%
		QC	(9.65 sec)	100%
rhino (F)	NullPointerException	Zest	(18.6 sec)	100%
		AFL	✗	0%
		QC	(9.85 sec)	100%
rhino (H)	ClassCastException	Zest	(245.18 sec)	85%
		AFL	✗	0%
		QC	(362.43 sec)	35%
rhino (J)	VerifyError	Zest	(94.75 sec)	100%
		AFL	✗	0%
		QC	(229.5 sec)	80%

Table 1: Average time to discovery (TTD) and Reliability (Rel.)—the percentage of runs on which the bug was found—for bugs found by each technique during our experiments. Bugs are deduplicated by benchmark and exception type. Dash "-" indicates bug was not found.

	RLCheck		QuickCheck		Zest	
Bug ID	TTD	Rel.	TTD	Rel.	TTD	Rel.
Ant, (#1)	41s	50%	178s	10%	123s	90%
Closure, (#2)	1s	100%	1.2s	100%	23s	60%
Rhino, (#3)	95s	90%	62s	70%	276s	10%
Rhino, (#4)	11s	100%	1s	100%	30s	100%
Rhino, (#5)	-	-	3s	100%	80s	100%
Rhino, (#6)	-	-	96s	20%	-	-

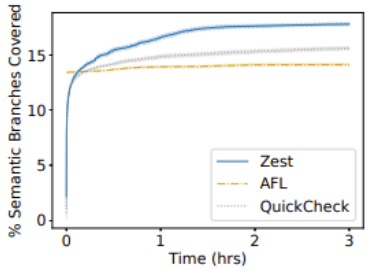
A Note on Performance

- When we run QuickCheck for 3 hours, should we run it with coverage feedback?
- We did in Zest
- Is this a reasonable decision? Why/why not?

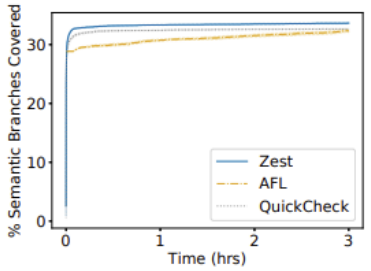


(a) maven

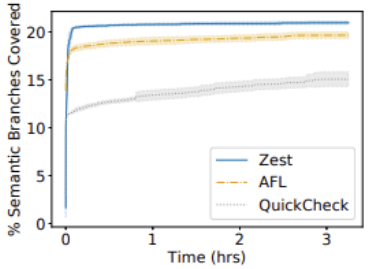
(b) ant



(c) closure



(d) rhino



(e) bcel

Parametric Generators: Convergent Evolution

- We mention crowbar as also doing this, other papers
- libFuzzer "FuzzedDataProvider" is like byte-based random
- First dinner at Shonan in 2019:
  - What are you presenting about?

Assorted Q's

- J Generator any good?
- Runtime: Why 3 hours?
- Benchmarks: why those?
- Generator quality?
- Where are the roperties?