Zest: Validity Fuzzing and Parametric Generators for Effective Random Testing

Preprint · November 2018

CITATIONS
0

READS
269

5 authors, including:

Koushik Sen
Daffodil International University
161 PUBLICATIONS **12,090** CITATIONS
SEE PROFILE

Mike Papadakis
University of Luxembourg
107 PUBLICATIONS **2,547** CITATIONS
SEE PROFILE

Yves Le Traon
University of Luxembourg
391 PUBLICATIONS **10,703** CITATIONS
SEE PROFILE

Zest: Validity Fuzzing and Parametric Generators for Effective Random Testing

Rohan Padhye*, Caroline Lemieux*, Koushik Sen*, Mike Papadakis†, Yves Le Traon†
* University of California, Berkeley
† University of Luxembourg
{rohanpadhye, clemieux, ksen}@cs.berkeley.edu, {michail.papadakis, yves.letraon}@uni.lu

***Abstract*—Programs expecting structured inputs often consist of both a *syntactic analysis stage* in which raw input is parsed into an internal data structure and a *semantic analysis stage* which conducts checks on this data structure and executes the core logic of the program. Existing random testing methodologies, like coverage-guided fuzzing (CGF) and generator-based fuzzing, tend to produce inputs that are rejected early in one of these two stages. We propose Zest, a random testing methodology that effectively explores the semantic analysis stages of such programs. Zest combines two key innovations to achieve this. First, we introduce *validity fuzzing*, which biases CGF towards generating semantically valid inputs. Second, we introduce *parametric generators*, which convert input from a simple parameter domain, such as a sequence of numbers, into a more structured domain, such as syntactically valid XML. These generators enable parameter-level mutations to map to structural mutations in syntactically valid test inputs. We implement Zest in Java and evaluate it against AFL and QuickCheck, popular CGF and generator-based fuzzing tools, on six real-world benchmarks: Apache Maven, Ant, and BCEL, ScalaChess, the Google Closure compiler, and Mozilla Rhino. We find that Zest achieves the highest coverage of the semantic analysis stage for five of these benchmarks. Further, we find 18 new bugs across the benchmarks, including 7 bugs that are uniquely found by Zest.**

I. Introduction

Programs expecting complex structured inputs often process their inputs and convert them into suitable data-structures before invoking the actual functionality of the program. For example, a build system such as Apache Maven first parses its input as an XML document and checks its conformance to a schema before invoking the actual build functionality. Compilers, PDF renderers, image processors and viewers, and various other programs whose inputs are XML files or JSON documents, all follow this same check-then-run pattern.

In general, such programs have an input processing pipeline consisting of two broad stages: a syntax parser and a semantic analyzer. We illustrate the flow of inputs through these stages in Figure 1. The syntax parsing stage translates the raw input into an internal data structure that can easily be processed (e.g. an abstract syntax tree) by the rest of the program. The semantic analysis stage checks if an input satisfies certain semantic constraints (e.g. checks if an XML input conforms to a specific schema), and performs further transformations to convert the input into a data structure that can be processed by the rest of the program. Inputs may be rejected by either stage if they are *syntactically* or *semantically invalid*. If an input passes both stages, then the input is considered *valid*.

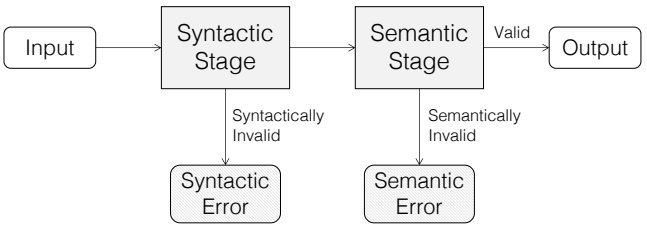

Fig. 1: Typical pipeline through programs expecting structured inputs. Inputs can either be syntactically invalid, semantically invalid, or valid and giving measurable output.

In this paper, we present Zest, a technique for generating test inputs that exercise the semantic analysis stages of programs, using a random testing methodology biased towards producing valid inputs.

Random testing has been successful in testing many different types of programs, from simple command-line tools to various components in web servers and browsers [1]–[8]. A popular variant of random testing is *coverage-guided mutational fuzzing* (CGF), in which the program under test is instrumented to provide feedback about the code coverage attained by the program when executing a given input. State-of-the-art CGF tools such as AFL [4] and libFuzzer [7] have found numerous critical bugs and security vulnerabilities in widely-used software systems. However, most of the bugs found by these CGF tools lie in the syntax-parsing stage of programs. These tools are usually ineffective at reaching the so-called *deep* states of programs, which perform various semantic analyses and transformations.

Another variant of random testing, called *generator-based fuzzing*, aims to exercise the core functionality of the program under test. The main idea behind generator-based fuzzing is to handcraft a probabilistic program which randomly generates syntactically valid inputs. QuickCheck [2], a popular generator-based fuzzing tool, uses a type-specific generator to check that a given assertion on inputs of that type likely holds. Grammar-based fuzzing [9]–[11] is another generator-based approach which generates syntactically valid inputs by probabilistically choosing production rules from a context-free grammar. Generator-based fuzzing tools can effectively test programs with mostly syntactic requirements on inputs.

However, it is much harder to use them to test code deep in the semantic analysis stage of programs and beyond. This is because in practice, it is very difficult to design a generator that produces inputs (1) satisfying complex semantic invariants and (2) that exercise a variety of code paths in the program under test. For example, the developers of CSmith [12], a tool that generates random C programs for testing compilers, spent several years manually tuning the generator to reliably produce valid C programs and to maximize code coverage in compilers.

Our proposed technique, Zest, consists of two sub-techniques which form the key contributions of this paper. First, we introduce *validity fuzzing*, an algorithm for biasing coverage-guided fuzzing towards generating semantically valid inputs. We call this component Zest_V. Second, we build upon Zest_V by leveraging ideas from generator-based fuzzing. We introduce *parametric generators*, which adapt existing probabilistic generators to be amenable to mutational fuzzing. We can then use validity fuzzing to bias the parametric generator towards producing semantically valid inputs. We call this combined technique Zest_VG.

Our hypothesis is that the bias towards generating inputs that are semantically valid will enable increased code coverage in the semantic analysis stages of programs, as well as aid in the discovery of bugs that are hard to find with existing random testing tools. Unlike some techniques that bias fuzzing towards a specific part of the program such as AFLGo [13], Zest does not require any static analysis. Therefore, Zest can be used to test programs written in languages where building a whole-program call graph is infeasible.

We evaluate Zest_V and Zest_VG on six real-world Java benchmarks and compare them to the baseline CGF and generator-based tools: AFL and QuickCheck, respectively. Our results show that the Zest_VG technique achieves the highest code coverage in the semantic analysis stage in five of our six benchmarks; the only exception is the benchmark with the simplest input format, where AFL performs best. Further, we find 18 new bugs across all the benchmarks during our evaluation. Zest_V and Zest_VG complement each other: together they discover all the 18 new bugs we find, including 7 unique bugs that are not found by either baseline technique.

To summarize, we make the following contributions:

- We present *validity fuzzing*, an algorithm for biasing coverage-guided mutational fuzzing towards the production of semantically valid inputs.
- We introduce *parametric generators*, which adapt existing probabilistic generators to be amenable to mutational validity fuzzing.
- We integrate these techniques, together called Zest, into the Java-based JQF platform, which we have made available as an open-source tool: https://github.com/rohanpadhye/jqf.
- We evaluate Zest against AFL and QuickCheck on six real-world Java benchmarks to compare their effectiveness in (1) generating semantically valid inputs, (2) covering code in the semantic analysis stages of test programs, and (3) discovering correctness bugs.

II. Background

In this section, we describe coverage-guided mutational fuzzing and generator-based fuzzing. We describe Zest in Section III.

A. Coverage-Guided Mutational Fuzzing

Algorithm 1 describes coverage-guided mutational fuzzing (CGF). The algorithm maintains a set $\mathcal{S}$ of important test inputs, which are used as candidates for future mutations. $\mathcal{S}$ is initialized with a user-provided set of initial seed inputs I (Line 1). The algorithm repeatedly cycles through the elements of $\mathcal{S}$ (Line 4), each time picking an input and using it as the base input from which to generate new inputs via mutation. The number of inputs to generate from a given input is determined by some function NUMCANDIDATES (Line 5), an implementation-specific heuristic. An input is generated by applying one or more random mutation operations on the base input (Line 6). These mutations may include operations that combine subsets of other inputs in $\mathcal{S}$. The given program is then executed with each newly generated input (Line 7).

The key to the CGF algorithm is that instead of treating the test program as a black-box, the test program is instrumented to provide dynamic feedback in the form of code coverage for each run. The algorithm maintains in the variable *totalCoverage* the set of all *coverage points* (e.g. program branches) covered by the existing inputs. If the execution of a generated input leads to the discovery of new coverage points (Line 8), then this input is added to the set $\mathcal{S}$ for subsequent fuzzing (Line 9) and the newly covered coverage points are added to *totalCoverage*. (Line 10).

The whole process repeats until a time budget expires. Often this time budget is in the order of hours or days, allowing several millions of executions. Finally, CGF returns the generated corpus of test inputs $\mathcal{S}$ (Line 12). CGF can either be used as a technique to discover inputs that expose bugs—in the form of crashes or assertion failures—or to automatically generate a corpus of test inputs that cover various program features. Note that the set $\mathcal{S}$ only grows monotonically throughout the algorithm; inputs are never removed from this set. This is in contrast to classical genetic algorithms that maintain a fixed-size population of candidates.

A key limitation of existing CGF tools is that they work without any knowledge about the syntax of the input. State-of-the-art CGF tools [4], [7], [8] treat program inputs as sequences of bytes. This choice of representation also influences the design of their mutation operations, which include bit-flips, arithmetic operations on word-sized segments, setting random bytes to random or "interesting" values (e.g. 0, `MAX_INT`), cloning and deleting byte subsequences, and inserting strings from a user-provided dictionary at random offsets. These mutation operations are tailored towards exercising various code paths in programs that parse inputs with a compact syntax, such as parsers for media file formats, decompression routines, and network packet analyzers. CGF tools have been very successful in finding numerous memory-corruption bugs

Algorithm 1 Coverage-guided mutational fuzzing.

Given: program p, set of initial inputs I
Returns: a set of generated test inputs

$\mathcal{S} \leftarrow I$
$totalCoverage \leftarrow \emptyset$
repeat
  for *input* in $\mathcal{S}$ **do**
    for $1 \leq i \leq$ NUMCANDIDATES(*input*) **do**
      $candidate \leftarrow$ MUTATE(*input*, $\mathcal{S}$)
      $coverage \leftarrow$ RUN(p, *candidate*)
      if $coverage \not\subseteq totalCoverage$ **then**
        $\mathcal{S} \leftarrow \mathcal{S} \cup \{candidate\}$
        $totalCoverage \leftarrow totalCoverage \cup coverage$
until given time budget expires
return $\mathcal{S}$

(such as buffer overflow vulnerabilities) in such programs due to incorrect handling of unexpected inputs.

Unfortunately, this approach often fails to exercise the core functions of software that expects highly structured inputs. For example, when AFL [4] is applied on a program that processes XML input data, a typical input that it saves looks like:

```
<a b>ac&#84;a>
```

which mostly exercises code paths that deal with syntax errors—in this case an error-handling routine for unmatched start and end XML tags. It is very difficult to generate inputs that will exercise new, interesting code paths in the semantic analysis stage of a program via these low-level mutations. In our experiments with testing Apache Maven's processing of `pom.xml` files, we found that only about 0.03% of AFL-generated inputs were semantically valid, even though AFL was provided with both a reference `pom.xml` file as a seed input and a dictionary of Maven-specific tag names.

B. Generator-based fuzzing

Generator-based fuzzing tools [2], [9]–[12], [14] allow users to write a probabilistic generator for inputs that conform to a specific format expected by the program under test. Figure 2 shows a generator for XML documents in the `junit-quickcheck` [15] framework, which is a Java port of QuickCheck [2]. This generator is designed to return syntactically valid XML string via its `generate` API. It does this by using the XML DOM API in the JDK to construct an `XMLDocument` object, then serializing it. The root element of this document is constructed by invoking `genElement` (Line 5). The method `genElement` uses repeated calls to `random` to generate the element's tag name (Line 11), any embedded text (Lines 19, 20, and in `genString`), and the number of children (Line 14); it recursively calls `genElement` to generate each child node. We omitted code to generate attribute names and values, but it can be done analogously.

Figure 3 contains a sample test harness method `testProgram`, identified by the `@Test` annotation. This method expects a test input `xml` of type `String`; the `@From` annotation indicates that input strings should be generated randomly using the `generate` API provided by

```
1  class XMLGenerator implements Generator<String> {
2
3  @Override // For Generator<String>
4  public String generate(Random random) {
5    XMLElement root = genElement(random);
6    return (new XMLDocument(root)).toString();
7  }
8
9  private XMLElement genElement(Random random) {
10   // Generate element with random name
11   String name = genString(random);
12   XMLElement node = new XMLElement(name);
13   // Randomly generate child nodes
14   int n = random.nextInt(MAX_CHILDREN);
15   for (int i = 0; i < n; i++) {
16     node.appendChild(genElement(random));
17   }
18   // Maybe insert text inside element
19   if (random.nextBoolean()) {
20     node.addText(genString(random));
21   }
22   return node;
23 }
24
25 private String genString(Random random) {
26   // Randomly choose a length and characters
27   int len = random.nextInt(MAX_STRLEN);
28   String str = "";
29   for (int i = 0; i < len; i++) {
30     str += random.nextChar();
31   }
32   return str;
33 }
34
35 }
```

Fig. 2: A simplified XML document generator.

```
1  @Test
2  void testProgram(@From(XMLGenerator.class) String xml) {
3    Model model = readModel(xml);
4    assume(model != null); // validity
5    assert(runModel(model) == success);
6  }
7
8  private Model readModel(String input) {
9      try {
10         return ModelReader.readModel(input);
11     } catch (XMLParseException e) {
12         return null; // syntax error
13     } catch (ModelException e) {
14         return null; // semantic error
15     }
16 }
```

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

the class `XMLGenerator`. When invoked with a single test input, `testProgram` creates a domain-specific model of the input (Line 3). The model creation fails if the input XML document string does not meet certain syntactic and semantic requirements (Lines 12 and 14). If the model creation is successful, the check at Line 4 succeeds and the function runs the method `runModel` at Line 5 to test one of the core functionalities of the program under test.

An XML generator like the one shown in Figure 2 generates random, but syntactically valid, XML inputs. Such generators can be used to overcome the limitations of CGF that we described in Section II-A. However, the generated inputs need not be semantically valid. The inputs generated by the depicted

Algorithm 2 The validity fuzzing algorithm. Differences from traditional CGF highlighted in grey.

Given: program p, set of initial inputs I
Returns: a set of generated semantically valid test inputs

1: $\mathcal{S} \leftarrow I$
2: $\mathcal{V} \leftarrow \emptyset$
3: *totalCoverage* $\leftarrow \emptyset$
4: *validCoverage* $\leftarrow \emptyset$
5: **repeat**
6: **for** *input* in $\mathcal{S}$ **do**
7: **for** $1 \leq i \leq$ NUMCANDIDATES(*input*) **do**
8: *candidate* $\leftarrow$ MUTATE(*input,* $\mathcal{S}$)
9: *coverage*, *isValid* $\leftarrow$ RUN(p, *canditate*)
10: **if** *coverage* $\not\subseteq$ *totalCoverage* **then**
11: $\mathcal{S} \leftarrow \mathcal{S} \cup \{$*candidate*$\}$
12: *totalCoverage* $\leftarrow$ *totalCoverage* $\cup$ *coverage*
13: **if** *isValid* and *coverage* $\not\subseteq$ *validCoverage* **then**
14: $\mathcal{S} \leftarrow \mathcal{S} \cup \{$*candidate*$\}$
15: $\mathcal{V} \leftarrow \mathcal{V} \cup \{$*candidate*$\}$
16: *validCoverage* $\leftarrow$ *validCoverage* $\cup$ *coverage*
17: **until** given time budget expires
18: **return** $\mathcal{V}$

XML generator do not necessarily conform to the schema expected by the application. Writing generators that produce semantically valid inputs by construction is a challenging, manual effort.

When we tested Apache Maven's model reader for `pom.xml` files using a generator similar to Figure 2, we found that only 0.09% of the generated inputs were semantically valid. Moreover, even if the generator manages to generate semantically valid inputs, it may not generate inputs that exercise code deep in the semantic analysis stage of programs and beyond. In our experiments with Maven, the QuickCheck-based approach covers less than one-third of the branches in the semantic analysis stage than our proposed technique does.

III. Proposed Technique

Our approach, Zest, addresses the drawbacks of CGF and generator-based fuzzing using two key ideas. First, Zest modifies the CGF algorithm to keep track of code coverage achieved by semantically valid inputs in order to bias input generation towards semantically valid inputs; we call this technique *validity fuzzing*. We implement validity fuzzing on its own in Zest_V. Second, Zest converts a probabilistic input generator to an equivalent deterministic *parametric generator* suitable for coverage-guided validity fuzzing. We combine validity fuzzing and parametric generators in Zest_VG.

A. Validity Fuzzing

Algorithm 2 outlines the coverage-guided fuzzing algorithm used by Zest to bias input generation towards inputs that are syntactically and semantically valid. The algorithm extends a regular CGF algorithm (i.e. Algorithm 1) by keeping track of the coverage achieved by *valid inputs*.

Like Algorithm 1, Algorithm 2 is provided a program under test p and a set of initial inputs I, which is used to initialize the set $\mathcal{S}$ at Line 1. Additionally, a set $\mathcal{V}$ of valid inputs is initialized to the empty set at Line 2. Along with *totalCoverage*, which maintains the set of coverage points (e.g. branches) in p covered by all inputs in $\mathcal{S}$, Algorithm 2 also maintains a set of cumulative coverage points covered only by the (valid) inputs in $\mathcal{V}$. This set is maintained in the variable *validCoverage*, which is initialized at Line 4.

New inputs are generated using standard CGF mutations at Line 8. The program p is then executed on each input. During the execution, in addition to code-coverage feedback, the algorithm records in the variable *isValid* whether the input is valid or not. An input is considered valid if the execution of the program on the input does not terminate due to a syntax or semantic error.

As in Algorithm 1, a newly generated input is added to the set $\mathcal{S}$ at Lines 10–12 if it produces new code coverage. Additionally, if a generated input is *valid* and if it covers a coverage point that has not been exercised by *any previous valid input*, then the input is added to the sets $\mathcal{S}$ and $\mathcal{V}$. The cumulative valid coverage variable *validCoverage* is also updated accordingly at Lines 13–16. Adding the input to $\mathcal{S}$ under this new condition ensures that we keep mutating valid inputs that exercise the core program functionality. Our hypothesis is that this heuristic biases the search towards generating even more valid inputs and in turn increases code coverage in the semantic analysis stage.

As in Algorithm 1, the fuzzing loop repeats until a time budget expires. Finally, the algorithm returns the corpus of automatically generated valid inputs, $\mathcal{V}$.

We refer to a coverage-guided fuzzing technique that uses Algorithm 1 as Zest_V. In our experiments with Apache Maven, we find that Zest_V generates $18\times$ more semantically valid inputs on average than AFL in the same time budget. However, just like standard CGF, Zest_V also generates many inputs that are syntactically invalid, thus spending time stressing the parser instead of the semantic analysis. We address this issue by incorporating the syntactically-valid-by-construction approach of generator-based techniques, with the concept of parametric generators.

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>
```

This is because the mutation of the second number in the sequence only affected the choice of first character in the root node's tag name, which changed from "foo" to "Woo". The generated input x_2 is still syntactically valid, with "Woo" appearing both in the start and end tag delimiters, because the XML generator uses an internal DOM tree representation that is only serialized after the entire tree is generated.

As another example, suppose the fifth number in the sequence σ_1 is decremented by 1, producing the sequence σ_3: 3, 102, 111, 111, **1**, 3, …, 0, 0. Then the root node in the generated input x_3 will have one fewer child node:

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

This is because the choice of number of child nodes to generate at Line 14 in Figure 2 is mutated from 2 to 1. Since the remaining values in the sequence are the same, the first child node in x_3—<bar>Hello</bar>—is identical to the one in x_1. The parametric generator thus enables a structured mutation in the DOM tree, such as deleting a subtree, by simply changing one value in the parameter sequence. The parametric generator simply ignores any unused numbers towards the tail of the sequence leftover from a reduction in the size of the generated input.

We can convert any probabilistic generator (such as those written for QuickCheck-like frameworks) into a parametric generator by mocking the pseudo-random number source. Then, we can make a generator-based fuzzing tool amenable to coverage-guided mutational fuzzing by considering the *parameter sequence* as the input to mutate.

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}.

IV. Implementation

Zest is implemented on top of the open-source JQF platform, which provides a framework for specifying algorithms for feedback-directed fuzz testing of Java programs.

JQF instruments the program under test using the ASM bytecode-manipulation framework [16]. Java classes are instrumented on-the-fly as they are loaded by the JVM using a `javaagent`. The instrumentation allows JQF to observe code coverage events such as the execution of program branches and invocation of virtual method calls.

Fuzzing front-ends can plug-into JQF to provide mechanisms to generate inputs and register callbacks for listening to code coverage events. JQF ships with front-ends for AFL and QuickCheck, which we use in our evaluation in Section V. Since AFL [4] is an external tool written in C, JQF uses a proxy program to exchange program inputs and coverage feedback; the overhead of this inter-process communication is a negligible fraction of the test execution time. For QuickCheck, JQF uses the `junit-quickcheck` [15] library which is a port of QuickCheck on top of JUnit.

Zest is implemented as another fuzzing front-end. The *coverage points* used in Algorithm 2 are tuples of the form $\langle b, \lfloor \log_2(c) \rfloor \rangle$, where b is a program branch and c is number

of times b was executed for a given input. This allows Zest to save some generated inputs even if they do not increase branch coverage, as long as the execution count of some branch differs by orders of magnitude. This heuristic has been known to work well in existing CGF tools [17]. It is motivated by the observation that certain components in a test program may only be exercised if a preceding loop executes for a large or specific number of iterations.

Zest_{VG} converts the probabilistic generators written for `junit-quickcheck` to parametric generators by extending the library class `SourceOfRandomness`. The low-level method for generating pseudo-random bytes—`nextByte()`—is overridden to poll for bytes from a parameter sequence. Parameter sequences are extended (with actual pseudo-random values) or truncated as needed when more or fewer bytes are requested from the `SourceOfRandomness` for a given test execution, depending on what execution path the generator takes. Zest_{V} is implemented as a specialization of Zest_{VG} that uses the trivial generator for an array of bytes; this mimicks the behavior of traditional CGF tools that treat inputs as byte arrays.

V. Evaluation

In our evaluation, we compare Zest with baseline techniques AFL and `junit-quickcheck` (referred to as simply QuickCheck hereon) on their ability to (1) produce a test corpus of semantically valid inputs, (2) find bugs, and (3) cover the syntactic and semantic phases of programs.

We evaluate both Zest_{V}, which only implements validity fuzzing, as well as Zest_{VG}, which combines validity fuzzing with parametric generators. By comparing AFL and Zest_{V}, we evaluate the effectiveness of validity fuzzing over standard CGF. By comparing QuickCheck and Zest_{VG}, we evaluate the effectiveness of coverage-guided generator-based fuzzing over simply sampling a generator without feedback.

Benchmarks: We evaluate Zest and the baselines on the following set of real-world Java programs:

1) Apache Maven [18]: The test reads a `pom.xml` file and converts it into an internal `Model` structure. The test driver is similar to the one shown in Figure 3. An input `pom.xml` is considered valid if it is a valid XML document and if it conforms to the right schema.
2) Apache Ant [19]: Similar to Maven, this test reads a `build.xml` file and populates a `Project` object. An input is considered valid if it is a valid XML document and if it conforms to the schema expected by Ant.
3) Google Closure [20] statically optimizes JavaScript code. The test driver invokes the `Compiler.compile()` method with an input string which is expected to be a JavaScript program. This compiler is configured to perform `SIMPLE_OPTIMIZATIONS`, a list of standard passes such as constant folding, function inlining, and dead-code removal. An input is valid if Closure successfully returns a result without reporting an error.
4) Mozilla Rhino [21] compiles JavaScript to Java bytecode. The test driver calls `Context.compileString()` with a given input string. An input is valid if Rhino returns a compiled script.
5) ScalaChess [22] implements the rules of chess in Scala, and is the library that backs the popular `lichess.org` chess server. Our test driver invokes the `Forsyth` API to parse a chess-position representation in Forsyth-Edward Notation (FEN) [23], [24], and returns a `Situation` object only if the chess position is valid and playable. The syntax of FEN is much simpler than XML or JavaScript.
6) Apache's Bytecode Engineering Library (BCEL) [25] provides an API to parse, verify and manipulate Java bytecode. Our test driver takes as input a `.class` file as a byte-stream and uses the `Verifier` API to perform 3-pass verification of the class file according to the Java 8 specification [26]. An input is valid if BCEL finds no errors up to Pass 3A verification.

Seeds and dictionaries: For AFL and Zest_{V}, we provide one valid seed input for each benchmark. For Maven and Ant, we use reference `pom.xml` and `build.xml` files available from their respective documentations. For Closure and Rhino, we use a minified version of the popular React.JS [27] library. For Chess, we use the FEN string that represents the initial chess board position at the start of a game. For BCEL, we compile a simple Java program that prints "Hello World" to generate a seed `Hello.class` file. Additionally, AFL uses dictionary files to inject user-provided tokens as part of its mutation process. We provide a dictionary of Ant and Maven-specific XML tag names as well as a list of JavaScript keywords for their respective benchmarks. The initial parameter sequence for Zest_{VG} is randomly generated.

Generators: The Zest_{VG} and QuickCheck techniques use hand-written input generators. For Maven and Ant, we use an XML document generator similar to Figure 2. Strings for tags and attributes are generated by randomly choosing strings from a provided list of string literals that are scraped from class files in Maven and Ant. The generator is written in about 150 lines of Java code. For Closure and Rhino, we use a generator for a subset of JavaScript that contains about 300 lines of Java code. The generator produces strings that are syntactically valid JavaScript programs. The FEN generator for Chess, written in less than 100 lines of code, randomly picks piece types and colors for each square of the chess board, and randomly assigns castling rights and other metadata. Finally, the BCEL generator uses the BCEL API to generate `JavaClass` objects with randomly generated fields, attributes and methods with randomly generated bytecode instructions in about 500 lines of Java code. All generators were written by one of the authors of this paper in less than two hours each. Although these generators produce syntactically valid inputs, no effort was made to produce semantically valid inputs; doing so for a programming language can take years to perfect [12].

The generators, seeds, and dictionaries have been made publicly and anonymously available at https://goo.gl/GfLRzA.

Experimental setup: For our experiments we run each technique with a time budget of 3 hours for each benchmark, on a machine with an Intel(R) Core(TM) i7-5930K 3.50GHz

TABLE I: Statistics on the valid test inputs generated by each technique.

(a) Average number of semantically valid inputs generated by each technique, along with the percentage of the total number of generated inputs they represent. Higher is better.

	AFL	QuickCheck	Zest_V	Zest_VG
Maven	600 (0.03%)	9K (0.09%)	11.1K (0.2%)	1.1M (**16%**)
Ant	1.2K (0.05%)	37 (7^{-4}%)	5.6K (1.2%)	101K (**21%**)
Closure	1.8K (3.1%)	551K (24%)	91K (16%)	308K (**32%**)
Rhino	29K (7.1%)	1.9M (25%)	730K (20%)	945K (**35%**)
Chess	168K (2%)	70K (3%)	259K (**47%**)	249K (20%)
BCEL	426K (30%)	36K (0.17%)	889K (**32%**)	1.2M (11%)

(b) Average number of branches covered by semantically valid inputs. Higher is better. ($\pm x$) designates that x is the standard error of the mean.

	AFL	QuickCheck	Zest_V	Zest_VG
Maven	742 ($\pm$5)	1431 ($\pm$3)	737 ($\pm$0)	**2593** ($\pm$16)
Ant	2845 ($\pm$4)	3267 ($\pm$16)	2803 ($\pm$0)	**3614** ($\pm$18)
Closure	13,106 ($\pm$68)	14,396 ($\pm$37)	13,374 ($\pm$224)	**15,501** ($\pm$43)
Rhino	**6621** ($\pm$111)	6252 ($\pm$4)	6350 ($\pm$85)	6527 ($\pm$29)
Chess	**3309** ($\pm$29)	3063 ($\pm$5)	3195 ($\pm$108)	3088 ($\pm$3)
BCEL	1546 ($\pm$0)	1416 ($\pm$18)	1559 ($\pm$9)	**1561** ($\pm$43)

CPU and 16GB of RAM. All experiments are repeated 3 times to account for variation in non-deterministic choices in the fuzzing algorithms.

A. Semantically Valid Test Inputs Generated

Recall that the validity fuzzing algorithm used in Zest_V and Zest_VG biases fuzzing towards the generation of semantically valid test inputs covering a variety of behaviors. We compare the techniques on two fronts to evaluate whether it was successful in doing so. First, we look at the number of valid inputs generated by each technique, and what proportion of total inputs generated by each technique were valid. Second, we look at the branch coverage the valid inputs generated by each technique achieve.

To evaluate the first point, we keep track of the total number of valid and invalid (i.e., syntactically or semantically invalid) inputs generated by each tool during their fuzzing runs. Table Ia records the number of generated inputs that are valid as well as the percent of total generated inputs that this number corresponds to, averaged over the three 3-hour runs for each benchmark we consider.

To evaluate the second point, we look at the coverage achieved by the valid test inputs generated. Table Ib records the number of program branches exercised by valid inputs generated for each technique. The table shows the average number of branches hit by valid inputs over the 3 runs, with the standard errors written in parentheses.

From Table Ia, we see that for all benchmarks either Zest_V or Zest_VG generates the highest *proportion* of semantically valid test inputs. In addition, for four benchmarks, Zest_V or Zest_VG generate the highest *number* of valid inputs. QuickCheck generates a higher number of valid test inputs for Rhino and Closure in spite of having a lower proportion of valid inputs; this is because QuickCheck does not require the overhead of code-coverage feedback. However, this does not mean that the valid inputs generated by QuickCheck cover more functionality, as we discuss next.

From Table Ib, we see that Zest_VG achieves the highest average number of branches covered for all benchmarks except Rhino. By conducting a 2-tailed Student's t-test with $\alpha = 0.05$, i.e. 95% confidence, we conclude that Zest_VG achieves significantly more coverage compared to the other techniques on the Maven, Ant, and Closure benchmarks. AFL achieves higher coverage on Rhino and Chess, but this is not statistically significant; the confidence intervals overlap with those of Zest_VG and Zest_V respectively. In Rhino, most of the coverage gains that AFL achieves are in the syntax analysis stages only, as we will discuss in Section V-C.

The difference in the techniques' ability to generate a valid input corpus is particularly stark for the XML benchmarks (Maven, Ant), with Zest_VG generating a significantly higher number and proportion of valid inputs. These two benchmarks highlight the key advantage of Zest_VG over QuickCheck alone: Zest_VG's coverage-guided strategy generates many more semantically valid inputs.

The difference in number and proportion of valid inputs generated is less stark for the JavaScript benchmarks (Closure, Rhino). We suspect this is because it is relatively easier to generate small valid snippets of JavaScript code (e.g. `1 + 2`) than it is to generate well-formed XML documents that conform to a schema. Zest_VG leads in terms of valid inputs generated as well as the coverage achieved by valid inputs on Closure. We hypothesize this is because Closure is a complex benchmark. Closure's semantic analysis is much stricter than Rhino's—for example, Closure refuses to accept function declarations with duplicate argument names whereas Rhino compiles such programs without complaining. Further, Rhino simply performs straightforward AST-to-bytecode compilation while Closure performs complex transformations such as dead code elimination. We believe that Zest_VG's significantly higher coverage on the Closure benchmark is evidence of its advantage in exercising deep semantic analyses.

The Chess and BCEL benchmarks require inputs to be in a much more compact syntax: fixed-size strings for FEN and a binary format for `.class` files. In such programs, non-generator-based techniques such as AFL and Zest_V also perform well.

Overall, the results in Table I show that on our benchmarks, Zest's variants generate a larger proportion of valid inputs, corresponding to higher coverage achieved by valid inputs. The advantage of Zest_VG over Zest_V is more pronounced for programs taking highly structured inputs with strict semantic requirements, like matching a particular XML schema. Next, we evaluate whether these metrics reflect a better ability to test the programs, as illustrated by bug-finding ability.

TABLE II: New bugs found by each technique on the benchmarks. Each distinct bug is identified by a unique circled letter. Superscripts denote the number of repetitions (out of 3) that find the bug; higher is better.

	AFL	QuickCheck	$Zest_V$	$Zest_{VG}$	Unique
Maven	Ⓐ^{3}	-	Ⓐ^{2}	-	**1**
Ant	-	-	-	Ⓑ^{3}	**1**
Closure	Ⓒ^{2}	Ⓒ^{1}	Ⓒ^{3}	Ⓒ^{3} Ⓓ^{1}	**2**
Rhino	Ⓔ^{1}	Ⓕ^{3} Ⓖ^{3} Ⓙ^{3}	Ⓔ^{1} Ⓘ^{1}	Ⓕ^{3} Ⓖ^{3} Ⓙ^{3} Ⓗ^{2}	**6**
Chess	-	-	-	-	**0**
BCEL	Ⓚ^{1} Ⓛ^{1} Ⓜ^{1}	Ⓝ^{1} Ⓞ^{3}	Ⓚ^{3} Ⓛ^{3} Ⓜ^{3} Ⓟ^{3} Ⓠ^{3} Ⓡ^{3}	Ⓝ^{3} Ⓞ^{3}	**8**
Total	**6**	**6**	**10**	**9**	**18**

B. Bugs Found

The benchmark programs we tested are widely used, stable software systems, and we only used their release versions. However, during the course of running our experiments, the fuzzing tools discovered bugs in five of the six benchmarks. We reported these bugs to the project developers. $Zest_V$ and $Zest_{VG}$ detected 7 unique bugs not found by either baseline technique, on top of detecting all bugs the baselines detected.

We use the term *bug* here to refer to an instance of the test program throwing an undocumented run-time exception, such as a `NullPointerException` (NPE). Ideally, for any given input, the test program should either process it successfully or report it as invalid using a documented mechanism, such as throwing a checked `ParseException` on syntax errors.

Across all our experiments, the various fuzzing techniques generated over 14,000 buggy inputs that correspond to over 3,000 unique stack-traces. We manually triaged these inputs by filtering them based on exception type, message text, and source location, resulting in a corpus of what we believe are 18 unique bugs. These bugs are broken down by benchmark program and the technique(s) that found them in Table II. Filled circles correspond to bugs in the semantic analysis stage, while unfilled circles correspond to bugs in the syntax parser. Some bugs are found by multiple techniques. 6 of these bugs have been acknowledged by the project maintainers, whereas the rest are awaiting confirmation at the time of writing. We next provide some examples of the bugs we found.

In Maven, Ⓐ was an NPE thrown by the parser if it encounters an EOF without a newline while parsing a start tag (e.g. `"<Y"`). Since this only happens in syntactically invalid input, neither $Zest_{VG}$ nor QuickCheck encountered this bug.

In Ant, Ⓑ was an `IllegalStateException` encountered in a component that expected a particular attribute of the `<augment>` XML element to have been populated; this bug indicates an incomplete semantic analysis. $Zest_{VG}$ was the only technique that found Ⓑ, possibly because this bug is deep in the semantic analysis stage.

In Closure, all techniques discovered Ⓒ, an NPE in Closure's handling of arrow functions such as `"x => y"`. $Zest_{VG}$ uniquely discovered Ⓓ, an `IllegalStateException` in a semantic analysis component relating to processing declarations of variables. The bug is triggered when a new variable is declared after a `break` statement; $Zest_{VG}$ generated the following test-case:

```
while ((l_0)){
  while ((l_0)){
    if ((l_0)) { break;;var l_0;continue }
    { break;var l_0 }
  }
}
```

In Rhino, both AFL and $Zest_V$ discovered Ⓔ, an out-of-bounds (OOB) access in parsing, while the generator-based $Zest_{VG}$ and QuickCheck discovered assertion failures Ⓕ and Ⓖ in the code-generation logic, as well as the `VerifyError` Ⓙ, where a semantically valid input caused Rhino to compile the script into an invalid Java class file. This is effectively a correctness bug: Rhino accepts the JavaScript input but the JVM rejects the output that Rhino produces due to a bytecode-verification error. $Zest_{VG}$ uniquely discovers Ⓗ, a `ClassCastException` thrown during compilation when a node in Rhino's IR is incorrectly cast to an `ArrayLiteral` node. $Zest_V$ uniquely discovers Ⓘ, an assertion failure in the handling of escape sequences in string literals in the parser.

BCEL is a particularly interesting case. Since the input to the test driver is a binary class file with compact syntax, AFL and $Zest_V$ find a significant number of bugs without the use of generators. Bugs ⓀⓁⓂ raise NPEs and other run-time exceptions due to incorrect handling of binary fields, such as a negative array length, in the syntax parsing or some of BCEL's semantic verification passes. Bugs ⓅⓆⓇ are also found in the semantic verification passes, but they are found only by $Zest_V$ and not by AFL. These last 3 bugs demonstrate the importance of validity fuzzing in exploring deep semantic analyses. On the other hand, the generator-based techniques QuickCheck and $Zest_{VG}$ find a different class of semantic bugs. For example, Ⓝ is only triggered when the generated test input contains a bytecode instruction that invokes an interface method, where the interface is both implemented by the class whose code contains the instruction and is unresolved in the class-path.

Finally, the superscripts on each bug identifier in Table II represent the number of repetitions—out of the 3 repetitions that we conduct for each experiment—in which that technique discovered the bug. We see that $Zest_V$ and $Zest_{VG}$ generally have a much more likelihood of finding bugs than the other techniques—for example, $Zest_V$ finds ⓀⓁⓂ in all three of the repetitions, while AFL finds them in only one repetition.

In each of Ant, Closure and Rhino, $Zest_{VG}$ found at least one bug in the semantic analysis stage that no other technique could find. The union of $Zest_V$ and $Zest_{VG}$ exhaustively covers all bugs found during our evaluation, including 7 bugs found by neither baseline technique. Based on these experiences, we believe that $Zest_V$ and $Zest_{VG}$ complement each other on uncovering bugs in the syntax parsing and semantic analyses

TABLE III: Description of benchmarks with prefixes of class/package names corresponding to syntactic and semantic analyses.

Name	Version	Syntax Analysis Classes	Semantic Analysis Classes
Maven	3.5.2	`org/codehaus/plexus/util/xml`	`org/apache/maven/model`
Ant	1.10.2	`com/sun/org/apache/xerces`	`org/apache/tools/ant`
Closure	v20180204	`com/google/javascript/jscomp/parsing`	`com/google/javascript/jscomp/[A-Z]`
Rhino	1.7.8	`org/mozilla/javascript/Parser`	`org/mozilla/javascript/(optimizer|CodeGenerator)`
Chess	8.6.8	`chess/format`	`chess/(Board|Situation|variant)`
BCEL	6.2	`org/apache/bcel/classfile`	`org/apache/bcel/verifier`

TABLE IV: Average number of branches covered by all generated inputs in the syntactic and semantic analysis stages of programs. Higher is better. ($\pm x$) designates the standard error of the mean x.

		Syntactic	Semantic
Maven	AFL	1111 ($\pm$34)	643 ($\pm$15)
	QC	695 ($\pm$2)	501 ($\pm$5)
	Zest_V	921 ($\pm$20)	514 ($\pm$0)
	Zest_VG	**1187** ($\pm$1)	**1765** ($\pm$5)
Ant	AFL	**2418** ($\pm$15)	1033 ($\pm$3)
	QC	1820 ($\pm$3)	36 ($\pm$10)
	Zest_V	2280 ($\pm$11)	1047 ($\pm$1)
	Zest_VG	1899 ($\pm$16)	**1169** ($\pm$6)
Closure	AFL	4020 ($\pm$12)	9180 ($\pm$52)
	QC	3091 ($\pm$19)	10552 ($\pm$37)
	Zest_V	**4140** ($\pm$116)	9047 ($\pm$83)
	Zest_VG	3301 ($\pm$60)	**11509** ($\pm$73)
Rhino	AFL	**1709** ($\pm$33)	1627 ($\pm$5)
	QC	1232 ($\pm$6)	1613 ($\pm$2)
	Zest_V	1672 ($\pm$10)	1579 ($\pm$7)
	Zest_VG	1361 ($\pm$36)	**1655** ($\pm$7)
Chess	AFL	**449** ($\pm$1)	**630** ($\pm$0)
	QC	302 ($\pm$1)	612 ($\pm$2)
	Zest_V	415 ($\pm$11)	615 ($\pm$9)
	Zest_VG	302 ($\pm$2)	616 ($\pm$1)
BCEL	AFL	868 ($\pm$0)	767 ($\pm$0)
	QC	784 ($\pm$29)	841 ($\pm$35)
	Zest_V	900 ($\pm$8)	779 ($\pm$2)
	Zest_VG	**1009** ($\pm$4)	**1131** ($\pm$12)

stages of programs that are similar to our benchmarks.

C. Coverage of Semantic and Syntactic Components

Our main hypothesis for validity fuzzing was that biasing fuzzing towards generating valid inputs should lead to better code coverage in the semantic analysis stages of programs.

To evaluate this, we compute the code coverage by all generated inputs, both valid and invalid, and classify branches in the syntax analysis and semantic analysis phase. Each covered branch is classified by matching the fully-qualified class names of the class in which it is contained with the prefix patterns in Table III, which we isolated manually.

Table IV presents the results of this classification. Again, we present the mean and standard errors. At $\alpha = 0.05$, Zest_VG has significantly higher coverage in the *semantic analysis* stage on Maven, Ant, Closure and BCEL. In Rhino, Zest_VG's lead is not statistically significant. For Chess, AFL has the highest semantic code coverage, though the confidence intervals overlap with those of Zest_V. This benchmark shows that the parametric generators of Zest_VG may be overkill for programs that expect a simpler input syntax such as FEN. In the *syntax analysis* stage of programs, AFL and Zest_V have significantly higher coverage on the Ant, Closure, and Rhino benchmarks, with AFL having a significant lead on Ant. This is expected, since AFL and Zest_V generate many inputs corresponding to code paths that exercise syntax errors, while Zest_VG does not. Surprisingly, however, Zest_VG has significantly higher coverage in the syntax analysis stage of Maven and BCEL.

In summary, these results suggest that Zest_VG is complementary to byte-based CGF approaches, such as AFL and Zest_V, if the goal is to explore as many code paths as possible. In particular, Zest_VG exercises more of the semantic analysis stage of the program while AFL and Zest_V remain effective at exercising the syntactic parsing stages of the program.

VI. Threats to Validity

The advantages of Zest are particularly pronounced on inputs with more complex structures. Even though we targeted a variety of input structures of varying complexity, we cannot conclude that Zest will have advantages similar to those in Section V on all other structured inputs.

For the same program under test, the performance of Zest_VG may vary with different generators. We did not estimate how Zest_VG's performance depends on the quality of generators since we hand-wrote the simplest generators possible for our benchmarks. However, we believe our results—especially on Ant and Maven—suggest Zest_VG's ability to bias a simple generator towards deeper behavior makes its performance less reliant on generator quality than purely generative approaches, like QuickCheck.

Not all programs follow the pipeline outlined in Figure 1. Nonetheless we believe that Zest can be more generally applicable. For example, Zest_VG could also be useful in generating data structures with complex invariants. For programs with a two-stage pipeline, Zest_VG complements other tools such as AFL in generating a wide variety of code coverage in different components.

VII. Related Work

There are many works related to the automated test input generation problem, as surveyed by Anand et al. [28]. The majority of them focus on unit testing, specification/model-based testing, and security testing.

Several unit-test generation techniques focus on the generation of sequences of method calls. For example, Randoop [1] and EvoSuite [3] generate JUnit tests by incrementally trying

and combining sequences of calls. During the generation of sequence of calls, both Randoop and EvoSuite take some form of feedback into account. Randoop and Evosuite require no input from the user other than a specific class whose code is to be covered, then produce unit tests by directly invoking methods on the component classes. In contrast, Zest addresses the problem of generating raw input data that is structurally and semantically valid for testing core software components.

In the security community, several tools have been developed to improve the effectiveness of coverage-guided fuzzing in reaching deep program states [5], [13], [29], [30]. Of these, AFLGo and FairFuzz are the most similar to our work.

AFLGo [13] extends AFL to direct fuzzing towards generating inputs that exercise a program point of interest. AFLGo could potentially serve as an alternative to validity fuzzing (Algorithm 2) by directing fuzzing towards the validity-checking criteria in the test program. However, AFLGo relies on whole-program static analysis, using LLVM's link-time optimization (LTO) to construct a call graph that helps the tool find a *distance* to the fuzzing target location. In our ecosystem, this is not feasible. Constructing precise call graphs for Java is notoriously difficult due to wide-spread use of virtual methods and dynamic class loading [31], [32]. Zest is purely dynamic, and is therefore unaffected by such language features.

FairFuzz [30] modifies AFL to bias input generation towards branches that are rarely executed, but does not explicitly identify parts of the program that perform the core logic. In Zest, we bias input generation towards validity even if the semantic analysis stage is exercised frequently; our objective is to maximize code coverage in this stage.

QuickCheck [2] has been implemented in many languages and has successfully been used for property testing in various applications, such as telecommunication protocols and sensort networks [33], [34], as well as for fuzzing binary file formats [35]. Much like these techniques, Zest leverages generators to prune the irrelevant input space and direct the search towards the function of the program under test. However, in contrast to these methods, Zest uses code coverage to guide the search for test inputs that satisfy semantic constraints.

Targeted property-testing [36], [37] guides the input-generators used in property testing towards a testing objective by using techniques such as hill climbing and simulated annealing to test network topologies, routing trees and non-interference properties. Such techniques rely on the program under test to return numeric utility values to maximize. Zest is more general as it does not restrict semantic invariants of a test program in any way; therefore Zest relies on non-numeric search techniques such as coverage-guided mutational fuzzing.

Grammar-based fuzzing [9]–[11], [38], [39] is another related line of research that relies on grammar specifications to generate complex structured inputs. The underlying idea of these approaches is to restrict the input space to the syntactically valid input space, which is then explored either systematically or randomly. Godefroid et al. [38] translate a given grammar to a set of constraints that can be solved by a dedicated solver. These constraints can then be used to support black-box and white-box fuzzing. Beyene et al. [39] transform an input grammar into Java classes and use metaheuristic search techniques to guide test generation. CSmith [12] is a compiler testing tool that generates random C programs for differential testing. LangFuzz [14] generates random programs using a grammar and by recombining code fragments from a codebase. All these approaches fall under the category of generator-based fuzzing, but primarily focus on tuning the underlying generators rather than leveraging any code coverage feedback. Zest is not restricted to context-free grammars, and does not require any domain-specific tuning.

`libprotobuf-mutator` [40] combines structure-aware fuzzing with code coverage feedback. However, the tool uses protocol buffers [41] to specify input formats, which limits expressiveness. Zest can use arbitrary probabilistic programs as generators, and additionally guides fuzzing towards semantic validity.

Recently, there has been interest in generating input grammars from existing inputs, using machine learning [42] and language inference algorithms [43]. Similarly, DIFUZE [44] infers device driver interfaces from a running kernel to boostrap subsequent structured fuzzing. These techniques are complementary to Zest—the grammars generated by these techniques could be transformed into parametric generators for Zest_{VG}.

Unlike Zest, which uses coverage information as a heuristic for which inputs may yield new coverage under mutation, symbolic execution tools [38], [45]–[54] methodically explore the program under test by capturing path constraints and directly producing inputs which fit yet-unexplored path constraints. Symbolic execution can thus be used to precisely produce valid inputs exercising new behavior. The cost of this precision is that it can lead to the path explosion problem for larger programs, which causes scalability issues. Hybrid techniques that combine symbolic execution with coverage-guided fuzzing have also been proposed [55]–[57].

Finally, domain specific languages supporting the construction of test harnesses have also been developed. UDITA [58] is such a language for Java. It aims to assist test generation and supports several test generation and test filtering strategies. TSTL [59] is a scripting language for writing test harnesses. It includes tools to support, manage and analyze test generation that share a common library interface. We share the same idea of easy and simple definition of test templates—in our case, QuickCheck-like generators—as these approaches. However, Zest targets the test generation problem directly by automatically generating the inputs that achieve high coverage or expose faults, while UDITA and TSTL focus on the definition and management of the test inputs.

VIII. Conclusion

We have presented Zest, a technique for exploring deeper functional behavior of programs by biasing its input generation towards semantically valid test inputs. We evaluated one of the key contributions of Zest, validity fuzzing, both with and without our second contribution, parametric generators, in

Zest_V and Zest_VG, respectively. In our evaluation we found that parametric generators had a bigger impact on increasing the number of valid inputs for input formats with more complicated syntax (e.g., XML and JavaScript), and a more pronounced improvement in coverage for programs that perform complex semantic analyses (e.g. Closure). We found that Zest_V complements Zest_VG in terms of bug discovery, with Zest_V finding a superset of bugs discovered by AFL, and Zest_VG finding a superset of the bugs discovered by QuickCheck. Together, Zest_V and Zest_VG found 7 unique new bugs across the six benchmarks.

REFERENCES

[1] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, ser. OOPSLA '07, 2007.

[2] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '00. New York, NY, USA: ACM, 2000, pp. 268–279. [Online]. Available: http://doi.acm.org/10.1145/351240.351266

[3] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11, 2011.

[4] M. Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl, 2014, accessed August 21, 2018.

[5] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proceedings of the 2017 Network and Distributed System Security Symposium*, ser. NDSS '17, 2017.

[6] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990. [Online]. Available: http://doi.acm.org/10.1145/96267.96279

[7] L. C. Infrastructure, "libfuzzer," http://llvm.org/docs/LibFuzzer.html, 2016, accessed August 21, 2018.

[8] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, 2016.

[9] P. M. Maurer, "Generating test data with enhanced context-free grammars," *Ieee Software*, vol. 7, no. 4, pp. 50–55, 1990.

[10] D. Coppit and J. Lian, "Yagg: An easy-to-use generator for structured test inputs," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 356–359. [Online]. Available: http://doi.acm.org/10.1145/1101908.1101969

[11] E. G. Sirer and B. N. Bershad, "Using production grammars in software testing," in *Proceedings of the 2Nd Conference on Domain-specific Languages*, ser. DSL '99. New York, NY, USA: ACM, 1999, pp. 1–13. [Online]. Available: http://doi.acm.org/10.1145/331960.331965

[12] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and Understanding Bugs in C Compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11, 2011.

[13] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, 2017.

[14] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[15] P. Holser, "junit-quickcheck: Property-based testing, JUnit-style," http://pholser.github.io/junit-quickcheck, 2014.

[16] OW2 Consortium, "ObjectWeb ASM," https://asm.ow2.io, accessed August 21, 2018.

[17] M. Zalewski, "American fuzzy lop technical details," http://lcamtuf.coredump.cx/afl/technical_details.txt, 2014, accessed August 21, 2018.

[18] "Apache Maven," https://maven.apache.org, 2018, accessed August 24, 2018.

[19] "Apache Ant," https://ant.apache.org, 2018, accessed August 24, 2018.

[20] "Google Closure," https://developers.google.com/closure/compiler, 2018, accessed August 24, 2018.

[21] "Mozilla Rhino," https://github.com/mozilla/rhino, 2018, accessed August 24, 2018.

[22] "ScalaChess," https://github.com/ornicar/scalachess, 2018, accessed August 24, 2018.

[23] S. J. Edwards, "Portable Game Notation specification and implementation guide," http://www.thechessdrum.net/PGN_Reference.txt, 1994.

[24] "Forsyth-Edwards Notation," https://en.wikipedia.org/wiki/Forsyth-Edwards_Notation.

[25] "Apache Byte Code Engineering Library," https://commons.apache.org/proper/commons-bcel, 2018, accessed August 24, 2018.

[26] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 8 Edition*, 1st ed. Addison-Wesley Professional, 2014.

[27] "React.JS," https://reactjs.org, 2018, accessed August 24, 2018.

[28] S. Anand, E. K. Burke, T. Y. Chen, J. A. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013. [Online]. Available: https://doi.org/10.1016/j.jss.2013.02.061

[29] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, 2017.

[30] C. Lemieux and K. Sen, "FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18, 2018.

[31] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav, "Alias analysis for object-oriented programs," in *Aliasing in Object-Oriented Programming*, D. Clarke, J. Noble, and T. Wrigstad, Eds. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 196–232. [Online]. Available: http://dl.acm.org/citation.cfm?id=2554511.2554523

[32] M. Hirzel, A. Diwan, and M. Hind, "Pointer analysis in the presence of dynamic class loading," in *European Conference on Object-Oriented Programming*. Springer, 2004, pp. 96–122.

[33] T. Arts, J. Hughes, J. Johansson, and U. T. Wiger, "Testing telecoms software with quviq quickcheck," in *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, September 16, 2006*, 2006, pp. 2–10. [Online]. Available: http://doi.acm.org/10.1145/1159789.1159792

[34] L. Lampropoulos and K. Sagonas, "Automatic wsdl-guided test case generation for proper testing of web services," in *Proceedings 8th International Workshop on Automated Specification and Verification of Web Systems, WWV 2012, Stockholm, Sweden, 16th July 2012.*, 2012, pp. 3–16. [Online]. Available: https://doi.org/10.4204/EPTCS.98.3

[35] G. Grieco, M. Ceresa, A. Mista, and P. Buiras, "QuickFuzz: testing for fun and profit," *J. Syst. Softw.*, vol. 134, no. C, pp. 340–354, Dec. 2017. [Online]. Available: https://doi.org/10.1016/j.jss.2017.09.018

[36] A. Löscher and K. Sagonas, "Targeted property-based testing," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 46–56. [Online]. Available: http://doi.acm.org/10.1145/3092703.3092711

[37] A. Loscher and K. Sagonas, "Automating targeted property-based testing," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, vol. 00, Apr 2018, pp. 70–80. [Online]. Available: doi.ieeecomputersociety.org/10.1109/ICST.2018.00017

[38] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08, 2008.

[39] M. Beyene and J. H. Andrews, "Generating string test data for code coverage," in *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, 2012, pp. 270–279. [Online]. Available: https://doi.org/10.1109/ICST.2012.107

[40] K. Serebryany, V. Buka, and M. Morehouse, "Structure-aware fuzzing for Clang and LLVM with libprotobuf-mutator," 2017.

[41] Google, "Protocol buffers," https://developers.google.com/protocol-buffers, 2017.

[42] P. Godefroid, H. Peleg, and R. Singh, "Learn & fuzz: Machine learning for input fuzzing," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 50–59. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155573

[43] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 95–110. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062349

[44] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "DIFUZE: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 2123–2138. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134069

[45] L. A. Clarke, "A program testing system," in *Proc. of the 1976 annual conference*, 1976, pp. 488–491.

[46] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, pp. 385–394, July 1976.

[47] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05, 2005.

[48] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13, 2005.

[49] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, 2008.

[50] V. Chipounov, V. Kuznetsov, and G. Candea, "The s2e platform: Design, implementation, and applications," *ACM Transactions on Computer Systems.*, vol. 30, no. 1, p. 2, 2012.

[51] G. Li, I. Ghosh, and S. P. Rajan, "Klover: A symbolic execution and automatic test generation tool for c++ programs," in *CAV*, 2011, pp. 609–615.

[52] N. Tillmann and J. de Halleux, "Pex - white box test generation for .NET," in *Proceedings of Tests and Proofs*, Apr 2008.

[53] S. Anand, C. S. Păsăreanu, and W. Visser, "JPF-SE: a symbolic execution extension to Java PathFinder," in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2007.

[54] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 1083–1094.

[55] K. Böttinger and C. Eckert, "DeepFuzz: Triggering vulnerabilities deeply hidden in binaries," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, ser. DIMVA 2016. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 25–34. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_2

[56] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings of the 2016 Network and Distributed System Security Symposium*, ser. NDSS '16, 2016.

[57] S. Ognawala, T. Hutzelmann, E. Psallida, and A. Pretschner, "Improving function coverage with munch: A hybrid fuzzing and directed symbolic execution approach," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, ser. SAC '18. New York, NY, USA: ACM, 2018, pp. 1475–1482. [Online]. Available: http://doi.acm.org/10.1145/3167132.3167289

[58] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in UDITA," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, 2010, pp. 225–234. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806835

[59] J. Holmes, A. Groce, J. Pinto, P. Mittal, P. Azimi, K. Kellar, and J. O'Brien, "TSTL: the template scripting testing language," *STTT*, vol. 20, no. 1, pp. 57–78, 2018. [Online]. Available: https://doi.org/10.1007/s10009-016-0445-y