Learning to Fuzz: Application-Independent Fuzz Testing with Probabilistic, Generative Models of Input Data

Jibesh Patra, Michael Pradel

Technical Report TUD-CS-2016-14664

TU Darmstadt, Department of Computer Science

November, 2016

TECHNISCHE
UNIVERSITÄT
DARMSTADT

SOLA
SoftwareLab

Learning to Fuzz: Application-Independent Fuzz Testing with Probabilistic, Generative Models of Input Data

Jibesh Patra
Department of Computer Science
TU Darmstadt
jibesh.patra@gmail.com

Michael Pradel
Department of Computer Science
TU Darmstadt
michael@binaervarianz.de

Abstract

Fuzzing is a popular technique to create test inputs for software that processes structured data. It has been successfully applied in various domains, ranging from compilers and interpreters over program analyses to rendering engines, image manipulation tools, and word processors. Existing fuzz testing techniques are tailored for a particular purpose and rely on a carefully crafted model of the data to be generated. This paper presents *TreeFuzz*, a generic approach for generating structured data without an a priori known model. The key idea is to exploit a given corpus of example data to automatically infer probabilistic, generative models that create new data with properties similar to the corpus. To support a wide range of different properties, TreeFuzz is designed as a framework with an extensible set of techniques to infer generative models. We apply the idea to JavaScript programs and HTML documents and show that the approach generates mostly valid data for both of them: 96.3% of the generated JavaScript programs are syntactically valid and there are only 2.06 validation errors per kilobyte of generated HTML. The performance of both learning and generation scales linearly w.r.t. the size of the corpus. Using TreeFuzz-generated JavaScript programs for differential testing of JavaScript engines exposes various inconsistencies among browsers, including browser bugs and unimplemented language features.

1. Introduction

Testing complex programs requires complex input data. An effective approach for testing such programs is fuzz testing, i.e., to randomly generate input data. Fuzz testing has been successfully applied, e.g., to compilers [49], runtime engines [18, 23], refactoring engines [16], office applications [20], and web applications [45]. A common requirement for effective fuzz testing is to generate data that complies or almost complies with the input format expected by the program under test.

To generate (almost) valid input data, existing fuzz testing techniques essentially use two approaches. First, model-based approaches require a model of the input format, such as a probabilistic context-free grammar (PCFG). Csmith [49], FLAX [45], and LangFuzz [23] are examples of grammar-based approaches. Unfortunately, manually creating such a model is a time-consuming and strongly heuristic effort that cannot be easily adapted to other languages and even newer versions of the same language. Yang et al., who created the popular Csmith compiler testing tool, report that it took "substantial manual tuning of the 80 probabilities that govern Csmith's random choices" to "make the generated programs look right" [49]. Second, whitebox approaches analyze the program under test to generate input that triggers particular paths, e.g., based on symbolic execution. SAGE [20] and BuzzFuzz [17] are examples of whitebox fuzzing approaches. Unfortunately the assumption, that the tested program is available at input generation time, made by these approaches is not always given. e.g., when creating inputs for differential testing across multiple supposedly equivalent programs [33] or when fuzz testing remote web applications. Moreover, whitebox techniques often suffer from scalability issues.

This paper exploits the observation that for many input formats, there are various example inputs to learn from. Recent work on learning probabilistic models of code shows that models learned from many examples can be very powerful, e.g., for predicting missing parts of mostly complete data [8, 10, 22, 37, 38, 41, 43]. However, existing work does not use probabilistic, generative models to create completely new input data. Instead, they are tuned to fill in relatively small gaps in otherwise complete data, such as recommending an API call or an identifier name in an otherwise complete program.

This paper merges two streams of research, fuzz testing and learning probabilistic models of structured data, into a novel approach for learning how to test complex programs given examples of input data. We focus on input data that can be represented as a labeled, ordered tree, which covers many common formats, such as source code (represented as an AST), documents (PDF, ODF, HTML), and images (SVG, JPG). Our approach, called *TreeFuzz*, learns models of such input data by traversing each tree once while accumulating information. For each node and edge in the tree, TreeFuzz gathers facts that explain why the node or edge has a partic-

ular label and appears at a particular position in the tree. After having traversed all input data, the approach summarizes the gathered information into probabilistic models. Finally, based on the learned models, TreeFuzz generates new input data by creating trees in a depth-first manner.

Most existing works on probabilistic, generative models of structured data uses a single model that describes the data, such as n-gram-based models [22, 38] or graph-based models [37]. A key contribution of our work is to instead provide an extensible framework for expressing a wide range of models. Each model describes a particular aspect of the input format. We describe six models in this paper. For example, one of these models suggests child nodes based on parent nodes, similar to a PCFG. Another model suggests node labels in a way that enforces definition-use-like relationships between subtrees of a generated tree, a property that cannot be easily expressed by existing probabilistic, generative models. During generation, the approach reconciles models by ordering them and by letting one model refine the suggestions of previous models. The main benefits of this multi-model approach are that TreeFuzz considers different aspects of the input format and that extending TreeFuzz with additional models is straightforward.

The models supported by TreeFuzz are "single-traversal models", i.e., they are extracted during a single traversal of each tree, and they generate new trees in a single pass. The main benefit of this class of models is that they bound the time of learning and generation, leading to linear time complexity w.r.t. the number of examples to learn from and w.r.t. the number of generated trees. Furthermore, these models can express properties learned by n-gram-based models [22, 38], PCFGs, and conditioning function-based models [10, 41], as well as properties that cannot be expressed with existing approaches.

Our work is related to LangFuzz [23], which fuzz tests language implementations by recombining existing programs into new programs. TreeFuzz differs by learning a probabilistic, generative model of the input format of the application under test and by not requiring built-in knowledge about the format. Our work also relates to Deep3 [41], which learns a probabilistic model for predicting individual program elements. While learning, their approach synthesizes functions that become part of the model. To deal with the inherent complexity of synthesis, Deep3 must limit the search space for these functions and use aggressive sampling of input data. In contrast, TreeFuzz supports models that cannot be synthesized by Deep3 and has linear time complexity without sampling. Our work also differs by exploring a novel application, generating new data from scratch, whereas Deep3 predicts individual program elements in an otherwise complete program. We are not aware of any existing work that combines learned probabilistic models with fuzz testing.

As two examples of input formats that TreeFuzz is useful for, we apply the approach to JavaScript programs and HTML documents. As a concrete application of TreeFuzz-generated data, we use generated JavaScript programs for differential testing of web browsers.

Our evaluation assess the ability of TreeFuzz to generate valid input data, its performance and scalability, as well as its effectiveness for fuzz testing. The results show that, even though we do not provide a model of the target language, the approach generates input data that mostly complies with the expected input format. Specifically, given a corpus of less than 100 HTML documents, the approach creates HTML documents that have only 2.06 validation errors per generated kilobyte of HTML.[1] Given a corpus of 100,000 JavaScript programs, 96.3% of the created programs are syntactically valid and 14.4% of them execute without any runtime errors. Practically all of the generated data differs from the given example data. Using the TreeFuzz-generated JavaScript programs to fuzz test web browsers has revealed various inconsistencies, including browser bugs, unimplemented language features, and browser-specific behaviors that developers should be aware of.

In summary, this paper contributes the following:

- We present a novel language-independent, blackbox fuzz testing approach. It enables testing a variety of programs that expect structured input data.
- We are the first to use learned probabilistic language models for generating test input data.
- As a practical application, we show that TreeFuzz-generated data is efficient and effective at finding browser inconsistencies. We envision various other applications, such as testing compilers, interpreters, program analysis tools, image processors, and rendering engines.

2. Overview and Example

TreeFuzz consists of three phases. First, during the *learning* phase, the approach infers from a corpus of examples a set of probabilistic, generative models that encode properties of the input format. Second, during the *generation* phase, TreeFuzz creates new data based on the inferred models. Finally, the generated data serves as input for the *fuzz testing* phase.

As a running example, consider applying TreeFuzz to JavaScript programs and suppose that the corpus of examples consists only of the program in Figure 1(a)[2]. The approach represents data as a tree with labeled nodes and edges. Figure 1(b) shows a tree representation of the example program, which is the abstract syntax tree.

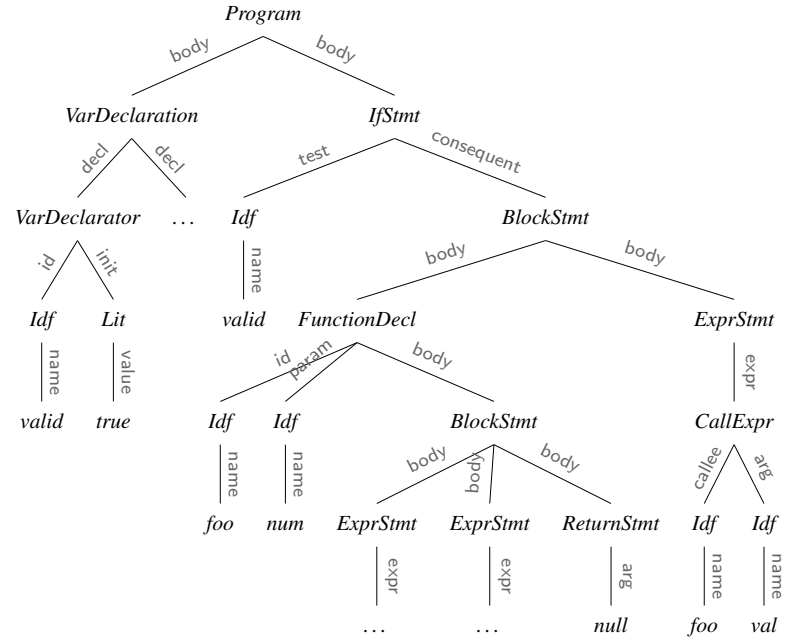
[1] The example documents are not perfect either: they contain 0.59 errors per kilobyte.

[2] For the evaluation (Section 6), we apply the approach to significantly larger corpuses.

(a) Example data from corpus:

```
var valid = true, val = 0;
if (valid) {
  function foo(num) {
    num = num + 1;
    valid = false;
    return;
  }
  foo(val);
}
```

(b) Tree representation of example data:



(c) Generated data:

```
// Program 1
var val = true, valid = true;
if (val) {
  foo(val);
  function foo(num) {
    return;
    return;
    val = num + 1;
  }
}
```

```
// Program 2
if (valid) {
  function foo(num) {
    return;
    valid = false;
    num = false;
  }
  foo(val);
}
var valid = 0, valid = 0;
```

```
// Program 3
if (valid) {
  foo(val);
  foo(val);
}
var valid = true, val = 0;
```

```
// Program 4
var valid = 0, valid = 0;
var valid = true, val = 0;
```

Figure 1. Corpus with a single example and new data generated from it. Parts of the abstract syntax tree have been abstracted for the sake of conciseness. *Idf* and *Lit* denote Identifier and Literal, respectively.

2.1 Learning

The learning phase of TreeFuzz traverses the tree of the example while inferring probabilistic, generative models of the input format. The models capture structural properties of the tree, which represent syntactic and semantic properties of the JavaScript language. For example, the approach infers that nodes labeled $Program$ have outgoing *body* edges and that these edges may lead to nodes labeled $VarDeclaration$ and $IfStmt$. Furthermore, the approach infers the probability of particular destination nodes. For example, for nodes labeled $BlockStmt$, an outgoing edge *body* leads to an $ExprStmt$ three out of five times. TreeFuzz infers similar properties for the rest of the tree, providing a basic model of the syntactic properties of the target language, similar to a PCFG. Existing grammar-based approaches use a pre-defined grammar, along with manually tuned probabilities to decide which grammar rules to expand.

TreeFuzz infers more complex properties in addition to the PCFG-like properties introduced above. For example, TreeFuzz considers the ancestors of nodes to find constraints about the context in which a particular node may occur. From the AST in Figure 1(b), the approach infers that nodes labeled $ReturnStmt$ always occur as descendants of a node $FunctionDecl$, i.e., the approach infers that return statements occur inside functions. Another inferred property considers repeatedly occurring subtrees. For example, the approach finds that the id edge of node $FunctionDecl$ and the $callee$ edge of node $CallExpr$ lead to identical subtrees $Idf \xrightarrow{name} foo$. If such a pattern occurs repeatedly in the corpus, TreeFuzz infers that $FunctionDecl$ and $CallExpr$ have a definition-use-like relation.

2.2 Generation

Based on the inferred models, TreeFuzz creates new trees. Figure 1(c) shows four examples of trees, pretty-printed as JavaScript programs. Tree generation starts in a top-down manner and nodes are iteratively expanded guided by the inferred models. For the example, an inferred model specifies that the root node of any tree is labeled $Program$, that $Program$ nodes have two outgoing edges, and that the children may be labeled $VarDeclaration$ or $IfStmt$. For this

reason, all four generated programs contain two statements, which are variable declarations or if statements. Generated programs have the same identifiers and literals as in the corpus because TreeFuzz infers the corresponding nodes.

To enforce the inferred constraint that return statements must appear within a function declaration, TreeFuzz only creates a $ReturnStmt$ node when the currently expanded node is a descendant of a $FunctionDecl$ node. As a result, the return statements in the first two programs of Figure 1(c) are within a function. Enforcing such constraints avoids syntax errors that TreeFuzz-generated programs would have otherwise. As an illustration of using complex properties encoded in the inferred model, recall the definition-use-like relation between $FuncDecl$ and $CallExpr$ that TreeFuzz infers. Suppose the approach generates the Idf subtree of a $CallExpr$ node. To select a label for the destination node of an edge $name$, the approach checks whether there already exists a $FunctionDecl$ node with a matching subtree, and if so, reuses the label of this subtree. As a result, most generated function calls in Figure 1(c) have a corresponding function declaration, and vice versa. Creating such relations avoids runtime errors during fuzz testing, e.g., due to undefined functions, that TreeFuzz-generated programs would have otherwise.

The models that TreeFuzz infers from a single example obviously overfit the example, and consequently, the generated programs do not use all features of the JavaScript language. The hypothesis of this work is that, given a large enough corpus of examples ("big code"), the approach learns a model that is general enough to create a variety of other valid examples that go beyond the corpus.

2.3 Fuzz Testing

Finally, the data generated by TreeFuzz is given as input to programs under test. For the running example, consider executing the generated programs in multiple browsers to compare their behavior. Executing the first program in Figure 1(c) exposes an inconsistency between Firefox 45 and Chrome 50. A bug in Firefox[3] causes the program to crash because the function `foo` declared in the `if` block does not get hoisted to the top of the block, which leads to a `ReferenceError` when calling it.

3. Learning and Generation

This section describes the first two phases of TreeFuzz: learning and generation. An important goal of TreeFuzz is to support different kinds of structured data, including programs written in arbitrary programming languages and structured file formats. A common format to represent such data are labeled, ordered trees, and we use this representation in TreeFuzz.

Definition 1. *A* labeled, ordered tree $t = (N, E)$ *consists of a set* N *of nodes and a set* E *of edges. Each node* $n \in N$ *and each edge* $e \in E$ *has a label. The function* $outgoing : N \to E \times ... \times E$ *maps each node to a tuple of outgoing edges. The function* $dest : E \to N$ *maps each edge to its destination node.*

For example, a labeled, ordered tree can represent the AST of a program, the DOM tree of a web page, a JSON file, an XML file, or a CSS file. Section 4 shows how to apply TreeFuzz to some of these kinds of structured data. In the remainder of the paper, we simply use the term "tree" instead of "labeled, ordered tree". To ease the presentation, we do not explicitly distinguish between a node and its label, or an edge and its label, if the meaning is clear from the context.

3.1 Extensible Learning and Generation Framework

To enable learning from a corpus of trees and generating new trees, TreeFuzz provides a generic framework that gets instantiated with an extensible set of techniques to infer probabilistic, generative models. We call these techniques *model extractors*. Each model extractor infers a particular kind of property from the given corpus and uses the inferred model to steer the generation of new trees. We currently have implemented six such model extractors, which Section 3.2 presents in detail.

3.1.1 Hooks

The TreeFuzz framework provides a set of hooks for implementing model extractors. The hooks are designed to support single-traversal models, i.e., the hooks are called during a single traversal of each example in the learning phase and during a single pass that creates new data during the generation phase. During the learning phase, TreeFuzz calls two hooks:

- $visitNode(node, context)$, which enables model extractors to visit each node of each tree in the corpus once, and
- $finalizeLearning()$, which enables model extractors to summarize knowledge extracted while visiting nodes.

During the generation phase, TreeFuzz calls three hooks:

- $startTree()$, which notifies model extractors that a new tree is going to be generated, enabling them to reset any tree-specific state,
- $pickNodeLabel(node, context, candidates)$, which asks model extractors to recommend a label for a newly created node,
- $pickEdgeLabel(node, context, candidates)$, which asks model extractors to recommend a label for the edge that is going to be generated next, and
- $havePickedNodeLabel(node, context)$, which notifies model extractors that a particular node label has been selected.

The *context* is the path of nodes and edges that lead from the tree's root node to the current node.

[3] Mozilla bug #585536

Algorithm 1 Learning phase.

Input: Set $\mathcal{T}$ of trees.
Output: Probabilistic, generative models.

1: **for all** $t \in \mathcal{T}$ **do**
2: $n \leftarrow root(t)$
3: $c \leftarrow$ initialize context with n
4: $\underline{visitNode}(n, c)$
5: **while** c is not empty **do**
6: **if** visited all $e \in outgoing(n)$ **then**
7: remove n from c
8: **else**
9: $e \leftarrow$ next not yet visited edge $\in outgoing(n)$
10: $n \leftarrow dest(e)$
11: expand c with e and n
12: $\underline{visitNode}(n, c)$
13: $\underline{finalizeLearning}()$

One important insight of this paper is that this simple API is sufficient to infer probabilistic models that enable generating trees suitable for effective fuzz testing.

3.1.2 Learning

To infer probabilistic, generative models that describe properties of the given set of trees, TreeFuzz traverses all trees while calling the hooks implemented by the model extractors. Algorithm 1 summarizes the learning phase. The algorithm traverses each tree in a top-down, depth-first manner and calls the $visitNode$ hook for each node. During the traversal, the algorithm maintains the context of the currently visited node. After visiting all trees, the algorithm calls $finalizeLearning$ to let model extractors summarize and store their extracted knowledge. Section 3.2 describes the model extractors in detail.

3.1.3 Generation

Based on the inferred models, which probabilistically describe properties of the trees in the corpus, TreeFuzz generates new trees that comply with these inferred properties. Algorithm 2 summarizes the generation phase of TreeFuzz. Trees are created in a top-down, depth-first manner while querying models about the labels a node should have, how many outgoing edges a node should have, and how to label these edges. The algorithm maintains a work list of nodes that need to be expanded. For each such node, the algorithm calls the $pickNodeLabel$ function of all models and repeatedly calls the $pickEdgeLabel$ function to determine the outgoing edges of the node. For each newly created outgoing edge, the algorithm creates an empty destination node and adds it to the work list. The algorithm has completed a tree when the work list becomes empty. Once a tree is completed, the algorithm adds it to the set $\mathcal{G}$ of generated trees.

Because models may continuously recommend to create additional outgoing edges, generating a tree may not terminate. To address this problem and to bound the size of gener-

Algorithm 2 Generation phase.

Input: Probabilistic, generative models.
Output: Set $\mathcal{G}$ of generated trees.

1: **while** $|\mathcal{G}| < maxTrees$ **do**
2: $\underline{startTree}()$
3: $n_{root} \leftarrow$ new node
4: $c \leftarrow$ initialize context with n_{root}
5: $N \leftarrow$ empty stack ▷ work list of nodes to expand
6: $N.push([n_{root}, c])$
7: **while** $|N| > 0$ **do**
8: $[n, c] \leftarrow N.pop()$
9: $\underline{pickNodeLabel}(n, c)$
10: $l_e \leftarrow \underline{pickEdgeLabel}(n, c)$
11: **while** $l_e \neq$ undefined **do**
12: add new edge with label l_e to $outgoing(n)$
13: $l_e \leftarrow \underline{pickEdgeLabel}(n, c)$
14: **for all** $e \in outgoing(n)$ **do**
15: $n_{dest} \leftarrow$ new node
16: $dest(e) \leftarrow n_{dest}$
17: $c_{dest} \leftarrow$ expand c with e and n_{dest}
18: insert $[n_{dest}, c_{dest}]$ into N
19: **if** $|reachableNodes(n_{root})| > \theta$ **then**
20: discard tree and continue with main loop
21: $\mathcal{G} \leftarrow \mathcal{G} \cup \{n_{root}\}$

ated trees, the algorithm checks (line 19) whether the current tree's total number of nodes exceeds a configurable threshold θ (default: 1,000) and discards the tree in this case.

The approach described so far provides a generic framework for inferring properties from a corpus of trees and for generating new trees based on these properties. The following section fills this generic framework with life by presenting a set of model extractors that are applicable across different kinds of data formats, such as JavaScript programs and HTML documents.

3.2 Model Extractors

To support a wide range of properties of data formats, TreeFuzz uses an extensible set of model extractors. Each model extractor implements the hooks from Section 3.1.1 to learn a model from the corpus and to make recommendations for generating new trees. The following explains six model extractors. They are sorted roughly by increasing conceptual complexity, starting from simple model extractors that learn PCFG-like properties and ending with model extractors that encode properties out of reach for PCFGs. Section 3.3 explains how TreeFuzz reconciles the recommendations made by different model extractors.

3.2.1 Determining the Root Node

Every generated tree needs a root node. This model extractor infers from the corpus which label root nodes typically have. During learning, the extractor builds a map $\mathcal{M}_{root}$ that as-

signs a label to the number of occurrences of the label in a root node. During generation, the model is used to recommend a label for the root node: When Algorithm 2 calls $pickNodeLabel$ with a context that only contains the current node (i.e., a root node), the model picks a label from the domain $dom(\mathcal{M}_{root})$ of the map, where the probability to pick a particular label n is proportional to $\mathcal{M}_{root}(n)$.

For the example in Figure 1(b), $\mathcal{M}_{root} = \{Program \mapsto 1\}$. When generating a new tree, the approach will recommend the label $Program$ for every root node.

3.2.2 Determining Outgoing Edges

The following model extractor infers the set of edges that a particular node label n should have, and uses this knowledge to suggest edge labels during the generation of trees. To this end, the approach maintains two maps. The map $\mathcal{M}_{edgeExists}$ assigns to an edge label e the probability that n has at least one outgoing edge e. The map $\mathcal{M}_{edgeNb}$ assigns to an edge label e a probability mass function that describes how many outgoing edges e the node n typically has.

Learning To construct these two maps, the model extractor implements the $visitNode$ hook and stores, for each visited node, the label of the node and the label of its outgoing edges, as well as how many outgoing edges with a particular label the node has. After all trees have been visited, the model extractor uses the $finalizeLearning$ hook to summarize the extracted facts into the maps $\mathcal{M}_{edgeExists}$ and $\mathcal{M}_{edgeNb}$.

For the example in Figure 1(b), the model extractor infers the following maps for node $BlockStmt$:

- $\mathcal{M}_{edgeExists} = \{body \mapsto 1.0\}$ because each $BlockStmt$ has at least one outgoing edge labeled "body".
- $\mathcal{M}_{edgeNb}$ maps $body$ to the probability mass function
$$f_{edgeNb}(k) = \begin{cases} 0.5 & \text{for } k = 2 \\ 0.5 & \text{for } k = 3 \\ 0 & \text{otherwise} \end{cases}$$
because 50% of all block statements have two outgoing $body$ edges and the other 50% have three outgoing $body$ edges.

Generation The inferred maps $\mathcal{M}_{edgeExists}$ and $\mathcal{M}_{edgeNb}$ are used by the $pickEdgeLabel$ hook to steer the generation of edges. At the first invocation of $pickEdgeLabel$ for a particular node, a list of outgoing edges are pre-computed based on the probabilities stored in these maps. At the first and all subsequent invocations of $pickEdgeLabel$ for a particular node, the model returns edge labels from this pre-computed list until each such label has been returned once. Afterwards, the model returns $undefined$ to indicate that no more edges should be created.

For the running example, suppose that the generation algorithm has created a node $BlockStmt$. When it calls $pickEdgeLabel$, the model will decide based on $\mathcal{M}_{edgeExists}$ that there needs to be at least one $body$ edge, and based on $\mathcal{M}_{edgeNb}$, that two such edges should be created. As a result, it will return $body$ for the first two invocations of $pickEdgeLabel$ and $undefined$ afterwards.

3.2.3 Parent-based Selection of Child Nodes

Each generated node needs a label. During learning, the following model extractor reads the incoming edge and the parent node from the context provided to $pickNodeLabel$ and keeps track of how often a node n is observed for a particular edge-parent pair. This information is then summarized using the $finalizeLearning$ hook into a map $\mathcal{M}_{child}$ that assigns a probability mass function f_{child} to each edge-parent pair.

For the example in Figure 1(b), the model extractor infers the following probability mass function for the edge-parent pair $(body, BlockStmt)$:
$$f_{child}(n) = \begin{cases} 0.6 & \text{if } n = ExprStmt \\ 0.2 & \text{if } n = FunctionDecl \\ 0.2 & \text{if } n = ReturnStmt \\ 0 & \text{otherwise} \end{cases}$$
During generation, the approach uses the inferred probabilities to suggest a label for a node based on the incoming edge and the parent of the node. For this purpose, the approach picks a node label according to the probability distribution described by f_{child}.

The properties learned by the previous three model extractors are similar to those encoded in a PCFG. Existing grammar-based fuzzing approaches, such as Csmith [49], hard code the knowledge that these model extractors infers. For example, the grammar encodes which outgoing edges a particular kind of node may have, and a set of manually tuned probabilities specifies how many statements a typical function body has, how many arguments a typical function call passes, and what kinds of statements typically occur within a block statement. Instead, TreeFuzz infers this knowledge from a corpus.

3.2.4 Ancestor-based Selection of Child Nodes

The model extractor in Section 3.2.3 infers the probability of a node label based on the immediate ancestor of the current node. While the immediate ancestor is a good default indicator for which node to create next, it may not provide enough context. For example, consider determining the destination node of the $value$ edge of a Lit node. Based on the parent only, the generator would choose among all literals observed in the corpus, ignoring the context in which a literal has been observed, such as whether it is part of a logical expression or an arithmetic expression.

To exploit such knowledge, we generalize the idea from Section 3.2.3 by increasing the amount of context to consider the k closest ancestor nodes and their connecting edges. We call the sequence of labels of these edges and nodes the *ancestor sequence*. For each such ancestor sequence, the model extractor infers a probability mass function, as described in Section 3.2.3, and uses this function during

generation to suggest labels for newly created nodes. In addition to the model extractor from Section 3.2.3, which is equivalent to $k = 1$, we also use a model extractor that considers the parent and grand-parent of the current node, i.e., $k = 2$. Supporting larger values of k is straightforward, but we have not found any need for a value of $k > 2$.

Since a grammar only encodes the immediate context of each node, existing grammar-based approaches cannot express such ancestor-based constraints. To avoid creating syntactically incorrect programs, Csmith uses built-in filters that encode syntactic constraints not obvious from a grammar. Instead TreeFuzz infers them from a corpus of examples.

3.2.5 Constraints on the Selection of Child Nodes

Tree structures often impose constraints on where in a tree a particular node may appear. For example, consider an AST node that represents a return statement. In the AST of a syntactically valid program, such a node appears only as a descendant of a node that represents a function. Enforcing such constraints while generating trees is challenging yet important to reduce the probability to generate invalid trees.

To address this challenge, this model extractor infers constraints of the following form:

Definition 2. *An* ancestor constraint $(n, \mathcal{N})$ *states that a node labeled* n *must have at least one ancestor from the set* $\mathcal{N} = \{n_{A1}, \ldots, n_{Ak}\}$.

Ancestor constraints are inferred in two steps. First, in the $visitNode$ hook, the approach stores for each node the set of labels of all ancestors of the node, as provided by the node's context. Second, in the $finalizeLearning$ hook, the approach iterates over all observed node labels and checks for each node label n whether all occurrences of n have at least one ancestor from a set $\mathcal{N}$ of node labels. If such a set $\mathcal{N}$ exists, then the approach infers a corresponding ancestor constraint. Otherwise, the approach adds n to the set $\mathcal{N}_{unconstr}$ of unconstrained node labels.

During generation, the approach uses the $pickNodeLabel$ hook to suggest a set of nodes that are valid in the current context. This set is the union of two sets. First, the set of all unconstrained nodes $\mathcal{N}_{unconstr}$, because these nodes are always valid. Second, the set of all nodes n that have an ancestor constraint $(n, \mathcal{N})$ where $\mathcal{N}$ has a non-empty intersection with the set of node labels in the current context.

3.2.6 Enforcing Repeated Subtrees

Complex trees sometimes contain repeated subtrees that refer to the same concept. For example, consider an AST that contains a function call and its matching function declaration, such as the two subtrees ending with foo in Figure 1(b). The Idf nodes of the call and the declaration have an identical subtree that specifies the name of the function. The following model extractor infers rules that specify which nodes of a tree are likely to share an identical subtree.

Definition 3. *An* identical subtree rule *states that if there exists a subtree* $n_A \xrightarrow{e_A} n_B \xrightarrow{e_b} x$ *in the tree, then there also exists a subtree* $n_D \xrightarrow{e_d} n_B \xrightarrow{n_b} y$ *in the same tree so that* $x = y$.

The notation $n \xrightarrow{e} n'$ denotes that a node labeled n has an outgoing edge labeled e whose destination is a node labeled n'. For each rule, the approach infers the support, i.e., how many instances of this rule have been observed, and the confidence, i.e., how likely the right-hand side of the rule holds given that the left-hand side of the rule holds.

For example, given the corpus of JavaScript programs that we use in the evaluation, TreeFuzz infers that

$$CallExpr \xrightarrow{callee} Idf \xrightarrow{name} x$$

implies

$$FunctionDecl \xrightarrow{id} Idf \xrightarrow{name} y$$

so that $x = y$ with support 59,146 and confidence 61.7%. This rule expresses that function calls are likely to have a corresponding function declaration with the same function name. The reasons for confidence being lower than 100% are that functions can also be declared through a function expression and that functions may be defined in other files.

Learning To infer identical subtree rules, the model extractors uses the $visitNode$ and $finalizeLearning$ hooks. When visiting a node n with context $\ldots \rightarrow n_A \xrightarrow{e_A} n_B \xrightarrow{e_B} n$, the approach stores the information that the suffix $n_B \xrightarrow{e_B} n$ has been observed with the prefix $\ldots \rightarrow n_A \xrightarrow{e_A}$. After visiting all trees, the $finalizeLearning$ hook summarizes the stored information into identical subtree rules by considering all suffixes that have been observed with more than one prefix. The approach increments the support of a rule for each node n for which the rule holds. To compute the confidence of a rule, the approach divides rule's support by the number of times the left-hand side of the rule has been observed.

Generation During generation, the approach uses the inferred identical subtree rules to suggest labels for nodes that are at positions x and y (as in Definition 3) of a rule. To this end, the approach maintains two maps. First, the map $\mathcal{M}_{pathToLabels}$ associates to a subtree $n_D \xrightarrow{e_D} n_B \xrightarrow{e_B}$ the set of labels x that have already been used to label the destination node of e_B. Second, the map $\mathcal{M}_{pathToLabelTodos}$ associates with a subtree $n_D \xrightarrow{e_D} n_B \xrightarrow{e_B}$ the set of labels that the generator still needs to assign to a destination node of e_B to comply with a identical subtree rule. Whenever the $havePickedNodeLabel$ hook is called, the approach checks if the current context matches any of the inferred rules. If the current node matches the left-hand side of a rule, then the approach decides with a probability equal to the rule's confidence that the right-hand side of the rule should also be true. If $\mathcal{M}_{pathToLabels}$ indicates that the right-hand side is not yet fulfilled, then the approach adds an entry to $\mathcal{M}_{pathToLabelTodos}$. Whenever the $pickNodelLabel$ hook

is called, the approach checks whether the current context matches an entry in $\mathcal{M}_{pathToLabelTodos}$. If it does, the approach fulfills the rule by suggesting the required label.

The last three model extractors show that single-traversal models can express rather complex rules and constraints that go beyond grammar-based approaches. Existing approaches, such as Csmith, hard code such constraints into their approach. For example, if Csmith generates a function call, it checks whether there is any matching function definition, and otherwise, generates such a function definition. TreeFuzz provides a general framework that allows for implementing a wide range of models beyond the six that we describe here.

3.3 Combining Multiple Model Extractors

When Algorithms 1 and 2 call a hook function, they call the function for each available model extractor. In particular, this means that multiple model extractors may propose different labels during the generation of trees. For example, suppose that while generating a tree, the generation algorithm must decide on the label of a newly created node. One model extractor, e.g., the one from Section 3.2.5, may restrict the set of available node labels to a subset of all nodes, and another model extractor, e.g., the one from Section 3.2.3 may pick one of the labels in the subset. Furthermore, when multiple model extractors provide contradicting suggestions, then the generation algorithm must decide on a single label.

To reconcile the suggestions by different model extractors, TreeFuzz requires to specify an order of precedence while querying the model extractors during generation. Each model extractor obtains the set of label candidates from the already queried extractors and returns another set of candidates which must be a subset of the input set, i.e., a model extractor can only select from the set of already pre-selected candidates. If, after querying all model extractors, the set of label candidates is non-empty, the generator randomly picks one of the candidates. If the set of candidates is empty, the generator falls back on a random default strategy, which sets node labels to the empty string and suggests to create another edge with an empty label with a configurable probability (default: 10%). During our evaluation, when using all model extractors described in this section, the set of candidates is practically never empty.

For the evaluation, we use the model extractors described in this section in the following order of precedence (high to low): Constraints on the selection child nodes, determining the root node, enforcing repeated subtrees, determining outgoing edges, ancestor-based selection of child nodes, and parent-based selection of child nodes.

4. Fuzz Testing

This section presents how to use TreeFuzz-generated data as inputs for fuzz testing. We consider two data formats: programs in the JavaScript programming language (Section 4.1) and documents in the web markup language HTML (Section 4.2).

4.1 JavaScript Programs

TreeFuzz generates ASTs of JavaScript programs by learning from the ASTs of a set of example programs. Generated programs may serve as test input for program analyses, refactoring tools, compilers, and other tools that process programs [23, 49]. We here use TreeFuzz-generated JavaScript programs for differential testing across multiple browsers, where the same program is executed in multiple browsers to detect inconsistencies among the browsers.

Our differential testing technique classifies programs into three categories: First, the behavior is *consistent* if there is no observable difference across browsers, which may be because the program either crashes in all browsers or does not crash all browsers. Second, the behavior is *inconsistent* if we observe a difference across browsers. This may be either because the program raises an exception in at least one browser but does not crash in another browser, or because the program crashes in all browsers but with different types of error, such as *TypeError* and *ReferenceError*. To compare errors with each other, we use the type of the thrown runtime error, as specified in the language specification. Finally, some programs are classified as *non-deterministic* because the behavior of different executions in a single browser differs, which we check by executing each program twice.

4.2 HTML Documents

As another input format, we apply TreeFuzz to the hypertext markup language HTML. Due to the popularity of HTML documents, there are various tools that require HTML documents as their input, such as browsers, text editors, and HTML processing tools. TreeFuzz generates inputs for these tools based on a corpus of example HTML documents, without requiring any explicitly given knowledge about the structure and content of HTML documents.

Since an HTML document consists of nested tags, there is a natural translation from such documents to labeled, ordered trees. We represent each tag as a node, where the label represents the tag name, such as $body$ and a. We represent nested tags through an edge between the parent and the child. The label of this edge is $childNode$ concatenated with the label of the destination node. The reason for copying the destination's label into the edge label is that otherwise, most edges would have the generic label $childNode$, which is not helpful in inferring the tree's structure. We represent attributes of tags, such as `id='foo'`, through child nodes with label $attribute$. These nodes have two outgoing edges, which point to the name and the value of the attribute, e.g., id and foo.

5. Implementation

We implement the approach into a framework with an extensible set of model extractors. The implementation can be

		minimum	median	maximum
HTML	file size (bytes)	39	77,604	703,327
	number of nodes	11	4,858	41,626
JS	file size (bytes)	3	2,438	7,241,063
	number of nodes	0	262	1,045,978

Table 1. HTML and JS corpuses used for learning.

easily instantiated for different input formats because most of the implementation of the framework and the model extractors is independent of the format. The JavaScript instantiation builds upon an existing parser [2] and code generator [1] and adds less than 300 lines of JavaScript code to the framework. The HTML instantiation builds upon an existing toolkit to parse and generate HTML documents [4] and adds less than 200 lines of JavaScript code to the framework. We implement differential testing as an HTTP server that sends JavaScript programs to client code running in different browsers, and that receives a summary of the programs' runtime behavior from these clients.

6. Evaluation

6.1 Experimental Setup

Corpus We use a corpus of 100,000 JavaScript files from GitHub [3]. For HTML, we visit the top 100 web sites (according to the Alexa ranking) and store the HTML files of their start page. Some sites appear multiple times in the top 100 list, e.g., `google.com` and `google.co.in`. We remove all but one instance of such duplicates and obtain a corpus of 79 unique HTML files. Table 1 summarizes the file size and the number of nodes in the tree representations of the corpuses.

Differential Cross-Browser Testing We instantiate the differential testing technique described in Section 4.1 with eight versions of the popular Firefox and Chrome browsers released over a period of four years: Firefox 17, 25.0.1, 33.1, 44, and Chrome 23, 31, 39, and 48.

All performance-related experiments are carried out on an Intel Core i7-4790 CPU (3.60GHz) machine with 32GB of memory running Ubuntu 14.04. We use Node.js 6.5 and provide it with 11GB of memory.

6.2 Validity of Generated Trees

TreeFuzz generates trees that are intended to comply with an input format without any a priori knowledge about this format. To assess how effective the approach is in achieving this goal, we measure the percentage of generated trees that pass language-specific validity checks.

JavaScript To measure whether a generated JavaScript program is valid, we pretty print it and parse it again. If the pretty printer rejects the tree or if the parser rejects the generated program, then we consider the program as syntactically invalid. 96.3% of 100,000 generated trees represent syntactically valid JavaScript programs. Furthermore, 14.4% of the syntactically valid programs execute without causing any runtime error.

HTML To measure the validity of generated HTML documents, we use the W3C markup validator [6]. In practice, most HTML pages are not fully compatible with the W3C standards and therefore cause validation errors. As a measure of how valid an HTML document is, we compute the number of validation errors per kilobyte of HTML.

The generated HTML documents have 2.06 validation errors per kilobyte. As a point of reference, the corpus documents contain 0.59 validation errors per kilobyte. That is, the generated documents have a slightly higher number of errors, but overall, represent mostly valid HTML. We conclude that TreeFuzz effectively generates HTML documents that mostly comply with W3C standards, without any a priori knowledge of HTML.

To the best of our knowledge, there is no existing approach based on a learned, probabilistic language models that generates entire programs with so few mistakes.

6.3 Influence of Corpus Size on Validity and Performance

Influence of Corpus Size on Validity To be effective, statistical learning approaches often need large amounts of training data. We evaluate the influence of the corpus size on the validity of TreeFuzz-generated programs. We measure the percentage of syntactically correct generated JavaScript programs while learning from a varying corpus size ranging from 10 to 100,000. We observe that the percentages vary between 96% and 98%, i.e., most generated programs are syntactically correct independent of the corpus size. We conclude from the results that the size of the corpus does not have a significant influence on the validity of the generated trees, suggesting that TreeFuzz is useful even when few examples are available.

Performance and scalability To enable TreeFuzz to learn from many examples and to generate large amounts of new data, the performance and scalability of the approach is crucial. Figure 2 shows how long the approach takes to learn depending on the size of the corpus, and how long it takes to generate 100 trees. The presented results are averages over three repetitions to account for performance variations. We observe that both learning and generation scales linearly with the size of the corpus. The main reason for obtaining linear scalability is that the approach focuses on single-traversal models, which scale well to larger corpuses.

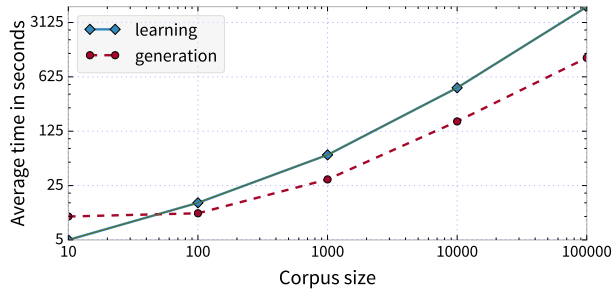


Figure 2. Learning and generation time based on varying corpus sizes. Both axes are log-scaled.

6.4 Effectiveness for Differential Testing

As an application of TreeFuzz-generated JavaScript programs, we evaluate the effectiveness for differential testing (Section 4.1) in two ways. First, we quantitatively assess to what extent the generated trees reveal inconsistencies. Second, we present a set of inconsistencies that we discovered during our experiments and discuss some of them in detail.

Quantitative Evaluation of Differential Testing The behavior of most programs (97.2%) is consistent across all engines, which is unsurprising because consistency is the intended behavior. For 0.22% of all programs, the behavior is non-deterministic, i.e., two executions in the same browser have different behaviors. Each of the remaining 2.5% of programs expose an inconsistency, i.e., achieves the ultimate goal of differential testing. Given the little time required to generate programs (Section 6.3), we conclude that TreeFuzz is effective at generating programs suitable for differential testing.

Qualitative Evaluation of Differential Testing To better understand the detected inconsistencies, we manually inspect a subset of all inconsistencies. Table 2 lists ten representative inconsistencies and associates them with three kinds of root causes. First, *browser bugs* are inconsistencies caused by a particular browser that does not implement the specified behavior. Second, *browser-specific behavior* are inconsistencies due to unspecified or non-standard features that some but not all browsers provide, or because the standards allow multiple different behaviors. Third, *missing revised-specification behavior* refers to inconsistencies due to features of not yet implemented revised specifications, such as ECMAScript 6 and DOM4. The examples listed in Table 2 show that TreeFuzz-generated JavaScript programs are effective at revealing different kinds of inconsistencies among browsers.

6.5 Comparison with Corpus and Other Approaches

We compare TreeFuzz to three alternatives approaches:

- The *simple, grammar-based* approach creates JavaScript programs based on built-in knowledge about the grammar of JavaScript's abstract syntax trees [5]. The approach generates programs by starting from the top-level AST node *Program* and by iteratively expanding nodes according to the grammar. When expanding a parent node by adding child nodes, each possible child node has the same probability of getting selected.
- *LangFuzz* [23], the state of the art approach that is closest to our work. Similar to TreeFuzz, it supports multiple languages and exploits a corpus of examples. In contrast to our work, LangFuzz requires built-in knowledge of the target language, such as which AST nodes represent identifiers and which built-in variables and keywords exist. For example, LangFuzz uses knowledge to adapt program fragments by modifying its identifier names.

  During our experiments, LangFuzz suffered from severe scalability problems when providing it with the full corpus of 100,000 programs. One reason is that the implementation keeps all programs in memory. Because of these problems, we provide it with 10,000 randomly sampled subset of of the corpus programs.

  The root cause of these memory issues is that LangFuzz combines fragments of existing programs with each other. Our approach avoids such problems by learning a probabilistic model of JavaScript code, instead of storing concrete code fragments.
- The *corpus-only* approach uses the 100,000 corpus programs as an input, i.e., no new programs get generated.

The simple, grammar-based approach suffers from two main limitations. First, it often fails to terminate because expanding a grammar rule often leads to the same grammar rule again. Second, most of the programs generated by the approach are repeated occurrences of very short programs, such as a program that defines only a single variable, or a even an empty program. Because of these limitations of the simple grammar-based approach, the rest of our comparison focuses on the other two approaches.

6.5.1 Comparison Based on Syntactical Differences

At first, we compare the approaches by syntactically comparing the programs that they provide. For this experiment, we format all programs consistently and remove all comments. We compare the programs generated by TreeFuzz and by LangFuzz with the programs in the corpus to assess whether any generated programs are syntactically equal to a corpus program. 241 of the 100,000 programs generated by LangFuzz are such duplicates, whereas only one of 100,000 TreeFuzz-generated programs is also present in the corpus. We conclude that TreeFuzz is effective at creating a large number of syntactically diverse programs.

The effectiveness of generated programs for fuzz testing partly depends on whether the programs are syntactically correct. The reason is that syntactically incorrect programs are typically rejected by an early phase of the JavaScript engine and therefore cannot reach any code beyond that phase. Figure 3 shows for each of the three approaches the percent-

ID	Inconsistent browsers	Description	Root cause
1	Firefox vs. Chrome	Mozilla bug #585536: Function declared in block statement should get hoisted to top of block.	Browser bug
2	Firefox 17 and 25 vs. others	Mozilla bug #597887: Calling `setTimeout` with an illegal argument causes runtime error.	Browser bug
3	Firefox 44 vs. others	Mozilla bug #1231139: `TypeError` is thrown even though it should be `SyntaxError`.	Browser bug
4	Firefox 17 and 25 vs. others	Mozilla bug #409444: The type of `window.constructor` is "object" in some browsers and "function" in others.	Browser bug
5	Firefox vs. Chrome	Only Firefox provides `window.content` property.	Browser-specific behavior
6	Firefox 44, Chrome 23, and Chrome 31 vs. others	Some browsers throw an exception when calling `scrollBy` without arguments.	Browser-specific behavior
7	Firefox vs. Chrome	`event` is a global variable in Chrome but not in Firefox.	Browser-specific behavior
8	Chrome 23 vs. others	Some browsers throw an exception when calling `setTimeout` without arguments.	Browser-specific behavior
9	Firefox 25–44 vs. others	Some browsers throw an exception when redirecting to a malformed URI.	Browser-specific behavior
10	Firefox 17–33 vs. others	Call of `Int8Array()` without mandatory `new` keyword, as required by ECMAScript 6.	Missing revised-specification behavior

Table 2. Examples of inconsistencies found through differential testing with TreeFuzz-generated programs.

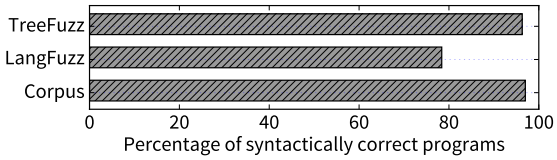


Figure 3. TreeFuzz compared to corpus and LangFuzz.

age of syntactically correct programs among all generated programs. For TreeFuzz and the corpus programs, the percentage is 96.3% and 97.0%, respectively. The fact that both values are similar confirms that TreeFuzz effectively learns from the given corpus. In contrast, only 78.4% of the programs generated by LangFuzz are syntactically correct.

6.5.2 Comparison Based on Differential Testing

To compare the programs generated by the different approaches beyond their syntax, we compare what kinds of inconsistencies the programs find when being used for differential testing. Since inspecting thousands of inconsistencies manually is practically infeasible, we assign inconsistencies to equivalence classes based on how an inconsistency manifests. These equivalence classes are an approximation of the actual root cause that triggers an inconsistency.

To compute the equivalence class of a program, we summarize the behavior of this program in a particular browser into a single string, such as "okay" for a non-crashing program and "ReferenceError" or "TypeError" for a crashing program. Based on these summaries, we compute a tuple $(b_1, \ldots, b_n)$ of strings for each program, where each b_i is the summary from a particular browser. Two inconsistencies belong to the same equivalence class if and only if they share the same tuple. For example, two programs that both throw

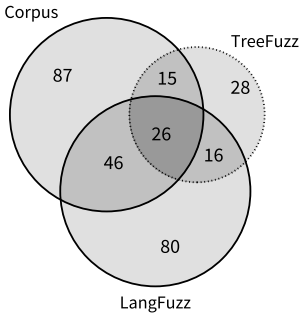


Figure 4. Equivalence classes of inconsistencies found by the three approaches.

a "TypeError" in all versions of Chrome but do not crash in any version of Firefox belong to the same equivalence class.

Figure 4 summarizes the results of the comparison. The figure shows for each approach how many equivalence classes of inconsistencies the approach detects, and how many equivalence classes are shared by multiple approaches. The results show that the three approaches are complementary to each other. Even though there is an overlap of 26 equivalence classes found by all three approaches, each individual approach contributes a set of otherwise missed inconsistencies. In particular, TreeFuzz detects 28 otherwise missed classes of inconsistencies.

Since Figure 4 is based on coarse-grained equivalence classes, the number of unique inconsistencies found by each approach is an underapproximation. To understand to what extent this abstraction underapproximates the number of unique root causes of inconsistencies that TreeFuzz finds, we manually inspect a sample of programs. Specifically, we randomly sample ten equivalence classes found by both TreeFuzz and an alternative approach, and inspect for each

class one program generated by TreeFuzz and one program generated by the other approach. The median of the number of programs in an equivalence class is one for corpus-only, TreeFuzz and two for LangFuzz. The goal of this manual inspection is to determine whether the inconsistencies exposed by the two programs are due to the same root cause.

In the pairs of programs inspected for the overlap between TreeFuzz and LangFuzz, we find for seven out of ten program pairs, the two have different root causes. Likewise, for the overlap between TreeFuzz and the corpus-only approach, the programs in eight out of ten pairs have different root causes. We conclude from these results that our equivalence classes are coarse-grained, i.e., Figure 4 is likely to be a strong underapproximation of the number of root causes exposed by the individual approaches.

7. Related work

Fuzz Testing Fuzz testing has been used to test UNIX utilities [34], compilers [49], runtime engines [18, 23], refactoring engines [16], other kinds of applications [20], and to find and exploit security vulnerabilities [28, 39, 45]. Blackbox fuzz testing either starts from existing data or generates new data based on a model that describes the required data format. For complex input formats, the model-based approach has the advantage that it avoids producing input data that is immediately rejected by the program. However, several authors mention the difficulties of creating an appropriate model for a particular target language [23, 39, 49], e.g., saying that "HTML is a good example of a complex file format for which it would be difficult to create a generator" [39]. Our work addresses this problem by inferring probabilistic, generative models of the data format. Whitebox fuzz testing analyzes the program under test to generate inputs that cover not yet tested paths, e.g., using symbolic execution [20, 35], concolic execution [19, 47], or taint analysis [17]. In contrast, TreeFuzz is independent of a particular program under test and therefore trivially scales to complex programs.

Corpus Analysis and Statistical Models Based on the observation that source code can be treated similarly to natural language documents [22], several statistical language models for programs have been proposed, e.g., based on n-grams [7, 9, 38], graphs [37, 42] and recurrent neural networks [43]. These models are useful for code completion [9, 22, 37, 38, 43], plagiarism detection [24], and to infer appropriate identifier names [7, 42]. Our work differs by learning probabilistic models that create entire programs and by being applicable to data beyond programs.

PHOG [10] and Deep3 [41] learn a model to predict how to complete existing data, e.g., for code completion. They pick the model depending on the context of the prediction and automatically synthesize a function that extracts this context. For performance reasons, PHOG and Deep3 limit the search space of the synthesis, e.g., by not synthesizing functions with loops. As a result, these approaches cannot express some of the model extractors supported by TreeFuzz, such as ancestor constraints (Section 3.2.5) and identical subtree rules (Section 3.2.6). Furthermore, TreeFuzz differs from PHOG and Deep3 by applying probabilistic models to fuzz testing, which requires to create data from scratch instead of predicting how to complete existing data.

Maddison and Tarlow propose a machine learning technique to generate "natural" source code [32]. TreeFuzz differs from their work by automatically inferring a generative model, instead of creating it by hand. Moreover, we evaluate the usefulness of generated programs and show that our approach applies to tree data other than programs. Our work also relates to other corpus-based analyses, e.g., to find anomalies that correspond to bugs [36], for code completion [12], to recommend API usages [50], for plagiarism detection [31, 46], and to find copy-paste bugs [29].

Testing Compilers and Runtime Engines Various efforts have been invested to test and validate compilers and runtime engines, starting from work in the 1960s [44], 1970s [21, 40], and 1990s [13, 33]. Several surveys [11, 25] provide an overview of older approaches. More recent work includes the Csmith approach for generating C programs for differential testing [49] and other random-based program generation techniques [30]. Instead of hard coding a model of the target language into the approach, TreeFuzz infers models from a corpus. Other work proposes oracles to determine whether a program exposes a bug in the compiler or execution engine [26, 27, 48] and on ranking generated programs [15]. Chen et al. empirically compare different compiler testing approaches [14]. Section 6.5 compares the JavaScript instantiation of TreeFuzz with a state of the art approach for generating JavaScript programs [23].

8. Conclusion

We present TreeFuzz, a language-independent, blackbox fuzz testing approach that generates tree-structured data. The core idea is to infer from a corpus of example data a set of probabilistic, generative models, which then create new data that has properties similar to the corpus. The approach does not require any a priori knowledge of the format of the generated data, but instead infers syntactic and semantic properties of the format. TreeFuzz supports an extensible set of single-pass models, enabling it to learn a wide range of properties of the data format. We apply the approach to two different data formats, a programming language and a markup language, and show that TreeFuzz generates data that is mostly valid and effective for detecting bugs through fuzz testing. Being easily applicable to any kind of tree-structured data, we believe that TreeFuzz can serve as a basis for various avenues for future work, e.g., generating input data for security testing and differential testing of program analysis tools.

References

[1] Escodegen ecmascript code generator. https://github.com/estools/escodegen. Accessed: 1-Nov-2016.

[2] Esprima parsing infrastructure for multipurpose analysis. http://esprima.org. Accessed: 1-Nov-2016.

[3] Learning from Big Code datasets. http://learnbigcode.github.io/datasets/. Accessed: 1-Nov-2016.

[4] parse5: whatwg html5 specification-compliant, fast and ready for production html parsing/serialization toolset for node.js. https://github.com/inikulin/parse5. Accessed: 1-Nov-2016.

[5] The ESTree specification. https://github.com/estree/estree/blob/master/es2015.md. Accessed: 1-Nov-2016.

[6] W3C markup validation service. https://validator.w3.org/. Accessed: 1-Nov-2016.

[7] M. Allamanis, E. T. Barr, C. Bird, and C. A. Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 281–293, 2014.

[8] M. Allamanis, E. T. Barr, C. Bird, and C. A. Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 38–49, 2015.

[9] M. Allamanis and C. A. Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pages 207–216, 2013.

[10] P. Bielik, V. Raychev, and M. T. Vechev. PHOG: probabilistic model for code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 2933–2942, 2016.

[11] A. Boujarwah and K. Saleh. Compiler test case generation methods: a survey and assessment. *Information and Software Technology*, 39(9):617 – 625, 1997.

[12] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 213–222. ACM, 2009.

[13] C. Burgess and M. Saidi. The automatic generation of test cases for optimizing fortran compilers. *Information and Software Technology*, 38(2):111 – 119, 1996.

[14] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie. An empirical comparison of compiler testing techniques. In *ICSE*, 2016.

[15] Y. Chen, A. Groce, C. Zhang, W. Wong, X. Fern, E. Eide, and J. Regehr. Taming compiler fuzzers. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 197–208, 2013.

[16] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 185–194. ACM, 2007.

[17] V. Ganesh, T. Leek, and M. C. Rinard. Taint-based directed whitebox fuzzing. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 474–484, 2009.

[18] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *PLDI*, volume 43, pages 206–215. ACM, 2008.

[19] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM, 2005.

[20] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[21] K. V. Hanford. Automatic generation of test cases. *IBM Syst. J.*, 9(4):242–257, Dec. 1970.

[22] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu. On the naturalness of software. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 837–847, 2012.

[23] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 38–38, Berkeley, CA, USA, 2012. USENIX Association.

[24] C. Hsiao, M. J. Cafarella, and S. Narayanasamy. Using web corpus statistics for program analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 49–65, 2014.

[25] A. S. Kossatchev and M. A. Posypkin. Survey of compiler testing methods. *Program. Comput. Softw.*, 31(1):10–19, Jan. 2005.

[26] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 216–226, New York, NY, USA, 2014. ACM.

[27] V. Le, C. Sun, and Z. Su. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 386–399, New York, NY, USA, 2015. ACM.

[28] S. Lekies, B. Stock, and M. Johns. 25 million flows later: large-scale detection of dom-based xss. In *ACM Conference on Computer and Communications Security*, pages 1193–1204, 2013.

[29] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: finding copy-paste and related bugs in large-scale software code. *Software Engineering, IEEE Transactions on*, 32(3):176–192, March 2006.

[30] C. Lindig. Random testing of c calling conventions. In *Sixth International Symposium on Automated and Analysis-*

Driven Debugging (AADEBUG), pages 3–11. ACM Press, Sept. 2005.

[31] C. Liu, C. Chen, J. Han, and P. S. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 872–881, New York, NY, USA, 2006. ACM.

[32] C. J. Maddison and D. Tarlow. Structured generative models of natural source code. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, pages 649–657, 2014.

[33] W. M. McKeeman. Differential testing for software. *DIGITAL TECHNICAL JOURNAL*, 10(1):100–107, 1998.

[34] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990.

[35] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 67–82, Berkeley, CA, USA, 2009. USENIX Association.

[36] M. Monperrus, M. Bruch, and M. Mezini. Detecting missing method calls in object-oriented software. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 2–25. Springer, 2010.

[37] A. T. Nguyen and T. N. Nguyen. Graph-based statistical language model for code. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 858–868, 2015.

[38] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A statistical semantic language model for source code. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 532–542, 2013.

[39] P. Oehlert. Violating assumptions with fuzzing. *IEEE Security & Privacy*, 3(2):58–62, 2005.

[40] P. Purdom. A sentence generator for testing parsers. *Bit Numerical Mathematics*, 12:366–375, 1972.

[41] V. Raychev, P. Bielik, and M. Vechev. Probabilistic model for code with decision trees. In *OOPSLA*, 2016.

[42] V. Raychev, M. T. Vechev, and A. Krause. Predicting program properties from "big code". In *Principles of Programming Languages (POPL)*, pages 111–124, 2015.

[43] V. Raychev, M. T. Vechev, and E. Yahav. Code completion with statistical language models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 44, 2014.

[44] R. L. Sauder. A general test data generator for cobol. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, AIEE-IRE '62 (Spring), pages 317–323, New York, NY, USA, 1962. ACM.

[45] P. Saxena, S. Hanna, P. Poosankam, and D. Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDSS*, 2010.

[46] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 76–85, New York, NY, USA, 2003. ACM.

[47] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 263–272. ACM, 2005.

[48] F. Sheridan. Practical testing of a c99 compiler using output comparison. *Softw. Pract. Exper.*, 37(14):1475–1488, Nov. 2007.

[49] 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, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294, 2011.

[50] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 318–343, 2009.