Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer

Suyoung Lee, HyungSeok Han, Sang Kil Cha, Sooel Son
School of Computing, KAIST

Abstract

JavaScript (JS) engine vulnerabilities pose significant security threats affecting billions of web browsers. While fuzzing is a prevalent technique for finding such vulnerabilities, there have been few studies that leverage the recent advances in neural network language models (NNLMs). In this paper, we present Montage, the first NNLM-guided fuzzer for finding JS engine vulnerabilities. The key aspect of our technique is to transform a JS abstract syntax tree (AST) into a sequence of AST subtrees that can directly train prevailing NNLMs. We demonstrate that Montage is capable of generating valid JS tests, and show that it outperforms previous studies in terms of finding vulnerabilities. Montage found 37 real-world bugs, including three CVEs, in the latest JS engines, demonstrating its efficacy in finding JS engine bugs.

1 Introduction

The memory safety of web browsers has emerged as a critical attack vector as they have become an integral part of everyday computing. Malicious websites, which conduct drive-by download attacks [48], have typically exploited memory corruption vulnerabilities of web browsers. Currently, an exploitable memory corruption vulnerability for a browser can cost 100,000 USD and sell for a million dollars if it is chained with a kernel exploit to remotely jailbreak iOS [59].

Among many components of web browsers, a JavaScript (JS) engine is of particular interest to attackers as its Turing-complete nature enables attackers to craft sophisticated exploits. One can easily allocate a series of heap chunks to perform heap spraying [49], write functions in JS to abstract away some exploitation logic [26], and even bypass the mitigation used in modern web browsers [35]. According to the National Vulnerability Database (NVD), 43% of the total vulnerabilities reported for Microsoft Edge and Google Chrome in 2017 were JS engine vulnerabilities.

Despite the increasing attention, there has been relatively little academic research on analyzing JS engine vulnerabilities compared to other studies seeking to find them [18, 24, 54]. LangFuzz [24] combines code fragments extracted from JS seed files to generate JS test inputs. GramFuzz and IFuzzer employ more or less the same approach [18, 54], but IFuzzer uses evolutionary guidance to improve the fuzzing effectiveness with genetic programming based on the feedback obtained by executing a target JS engine with produced inputs.

However, none of the existing approaches consider the relationship between code fragments for generating test inputs. In other words, they produce test inputs by simply combining fragments as long as JS grammars allow it. Thus, they do not determine which combination is likely to reveal vulnerabilities from the target JS engine. Are there any similar patterns between JS test inputs that trigger JS engine vulnerabilities? If so, can we leverage such patterns to drive fuzzers to find security vulnerabilities? These are the key questions that motivated our research.

We performed a preliminary study on JS engine vulnerabilities and observed two patterns. We observed that *a new security problem often arises from JS engine files that have been patched for a different bug.* We analyzed 50 CVEs assigned to ChakraCore, a JS engine used by Microsoft Edge. We found that 18% and 14% of the vulnerabilities were related to `GlobOpt.cpp` and `JavascriptArray.cpp`, respectively.

The second observation was that *JS test code that triggers new security vulnerabilities is often composed of code fragments that already exist in regression tests.* We collected 2,038 unique JS files from the ChakraCore regression test suite and 67 JS files that invoked the analyzed vulnerabilities. These two sets of files were disjoint. We sliced the AST of each JS file into AST subtrees of depth one, called *fragments*. We then computed the number of overlapping fragments between the two sets; we found that 95.9% of the fragments extracted from the 67 vulnerability-triggering JS files overlapped with the fragments extracted from the regression test suite (see §3).

Given these two observations, how do we perform fuzz testing to find JS engine vulnerabilities? For this research question, we propose the first approach that leverages a *neural network language model* (NNLM) to conduct fuzz testing

on a target JS engine. Our key idea is to mutate a given regression JS test by replacing its partial code with new code that the NNLM creates. Consider a regression JS test that invokes a patched functionality. We generate a JS test from this regression test while expecting to elicit a new potential bug that resides in the patched JS engine files, thus addressing the first observation. We also assemble existing code from regression test suites under the guidance of the NNLM when composing new partial code. This captures the second observation.

To manifest this idea, we designed and implemented Montage, a system for finding security vulnerabilities in JS engines. The system starts by transforming the AST of each JS test from a given regression test suite into the sequence of fragments. These fragment sequences become training instances over which the NNLM is trained. Therefore, the NNLM learns the relationships between fragments. Montage mutates a given JS test by reconstructing one of its subtrees as the trained NNLM guides.

Previous research focused on learning the relationships between PDF objects [16], characters [11, 32], and lexical tokens in the source code [22,40,43]. These language models addressed completing incorrect or missing tokens [40, 53], or assembling PDF objects [16]. Their methods are not directly applicable to generating valid JS tests, which requires modeling structural control flows and semantic data dependencies among JS lexical tokens. Liu *et al.* [32] stated their limitation in extracting general patterns from character-level training instances from C code, thus generating spurious tests.

Unlike these previous studies [11, 16], Montage uses fragments as building blocks. Each fragment encapsulates the structural relationships among nodes within an AST unit tree. The model is then trained to learn the relationships between such AST unit trees. Montage uses this model to assemble unit subtrees when mutating a given regression JS test. Thus, each generated JS test reflects the syntactic and semantic commonalities that exist in the regression test suite.

We evaluated Montage to find bugs in ChakraCore 1.4.1 and compared the number of found bugs against CodeAlchemist [20], jsfunfuzz [38], and IFuzzer [54]. We performed five fuzzing campaigns; each round ran for 72 hours. Montage found 133 bugs, including 15 security bugs. Among the found security bugs, Montage reported 9, 12, and 12 bugs that CodeAlchemist, jsfunfuzz, and IFuzzer did not find, respectively. This result demonstrates that Montage is able to find bugs that the state-of-the-art JS fuzzers are unable to find.

We measured the efficacy of the Montage language model against the random selection method with no language model, Markov-chain model, and the character/token-level recurrent neural network language model. Montage outperformed the other approaches in terms of finding unique bugs.

We further tested Montage to fuzz the latest versions of ChakraCore, JavaScriptCore, SpiderMonkey, and V8. Montage found 37 unique bugs, including three security bugs. 34 bugs were found from ChakraCore. The remaining two and one bugs were from JavaScriptCore and V8, respectively. Of these three security bugs, Montage discovered one from JavaScriptCore and the other two from ChakraCore. These results demonstrate the effectiveness of leveraging NNLMs in finding real-world JS engine bugs.

2 Background

2.1 Language Model

A language model is a probability distribution over sequences of words. It is essential for natural language processing (NLP) tasks, such as speech recognition, machine translation, and text generation. Traditionally, language models estimate the likelihood of a word sequence given its occurrence history in a training set.

An n-gram language model [8, 30] approximates this probability based on the occurrence history of the preceding $n-1$ words. Unfortunately, such count-based language models inherently suffer from the *data sparsity problem* [8], which causes them to yield poor predictions. The problem is mainly due to insufficient representative training instances. NNLMs address the data sparsity problem by representing words as a distributed vector representation, which is often called a *word embedding*, and using it as input into a neural network.

Bengio *et al.* [3] introduced the first NNLM, a feed-forward neural network (FNN) model. An FNN predicts the next word based on its preceding $n-1$ words, which is called a *history* or a *context* where n is a hyper parameter that represents the size of the word sequence [1, 3, 17]. In this NNLM setting, all words in a training set constitute a vocabulary V. Each word in V is mapped onto a feature vector. Therefore, a *context*, a word sequence, becomes the concatenation of each feature vector corresponding to its word. The model is then trained to output a conditional probability distribution of words in V for the next word from a given *context*.

Long short-term memory (LSTM). Unlike FNN language models, a recurrent neural network (RNN) is capable of predicting the next word from a history of preceding words of an arbitrary length because an RNN is capable of accumulating information over a long history of words. An LSTM model is a special kind of RNN; it is designed to capture long-term dependencies between words [14, 23]. Because a standard RNN suffers from the gradient vanishing/exploding problem [4], an LSTM model uses neural layers called gates to regulate information propagation and internal memory to update its training parameters over multiple time steps.

2.2 JS Engine Fuzzing

Fuzz testing is a form of dynamic software testing in which the program under test runs repeatedly with test inputs in order to discover bugs in the program. Fuzzing can be categorized into two types based on their input generation method-

ology [50]: mutational fuzzing and generational fuzzing. Mutational fuzzing [7, 44, 57, 58] alters given seeds to generate new test inputs, whereas generational fuzzing [19, 20, 24, 38] produces tests based on an input model, such as a grammar.

Since JS code is highly structured, randomly generated test inputs are likely to be rejected by JS engines. Therefore, it is common for JS engine fuzzers to employ a generational approach. One notable example is jsfunfuzz, a seminal JS engine fuzzer [38, 45]. It starts with a start symbol defined in a JS grammar and selects the next potential production in a random fashion until there are no remaining non-terminal symbols. CodeAlchemist [20] is another generational fuzzer that resort to the assembly constraints of its building blocks called code bricks to produce semantically valid JS code.

Most other JS engine fuzzers use both mutational and generational approaches. LangFuzz [24], GramFuzz [18], and IFuzzer [54] parse JS seeds with the JS grammar and construct a pool of code fragments, where a code fragment is a subtree of an AST. They combine code fragments in the pool to produce a new JS test input, but they also mutate given seeds to generate test inputs.

Although it does not aim to find security vulnerabilities, TreeFuzz [41] leverages a probabilistic context-free grammar (PCFG) to generate a test suite from given seeds. Similarly, Skyfire [56] infers a probabilistic context-sensitive grammar (PCSG) from given seeds and uses it to generate a well-distributed set of seeds. Both approaches apply probabilistic language models to generate JS testing inputs, but their design is too generic to find security vulnerabilities in JS engines. Unlike previous approaches, Montage is inspired by a systematic study of CVEs, i.e., previous JS engine vulnerabilities, and leverages an NNLM trained to learn syntactic and semantic commonalities between JS regression test suites.

3 Motivation

Can we find similarities between JS files that trigger security vulnerabilities? We answer this question by conducting a quantitative study of analyzing reported CVEs and corresponding proof of concept (PoC) exploits for ChakraCore [10]. We chose ChakraCore because its GitHub repository maintains well-documented commit logs describing whether a specific CVE is patched by a commit. This helps us identify which security vulnerability is related to a given PoC exploit and which source lines are affected by the vulnerability. Other JS engines, in contrast, have not provided an exact mapping between a code commit and a CVE.

Note that collecting PoC exploits is not straightforward because CVE reports typically do not carry any PoC exploits due to the potential risk of being abused. We manually collected CVEs as well as their PoC code from exploitDB, vulnerability blogs, and the ChakraCore GitHub repository. In total, we obtained 67 PoC exploits, each of which corresponds to a unique CVE. We further identified 50 of them where the corresponding vulnerabilities are fixed by a single commit. This means that we can map each of the 50 vulnerabilities to a set of affected source files. The earliest and the latest vulnerabilities in the collected set were patched in September 2016 and March 2018, respectively. In total, 77 files were patched owing to these vulnerabilities.

We found that nine out of the 50 vulnerabilities (18%) are related to the `GlobOpt.cpp` file, which mainly implements the just-in-time (JIT) compilation step. Seven of them (14%) have also contributed to patching the `JavascriptArray.cpp` file. Note that each file implements different functionalities of ChakraCore. In other words, different JS engine vulnerabilities often arise from a common file that implements the same functionalities, such as JIT optimization and JS arrays. For example, a patch for CVE-2018-0776 forces a deep copy of an array when the array is accessed via the function `arguments` property within a callee, thus avoiding a type confusion vulnerability. However, the patch was incomplete, still leaving other ways in which a shallow copy of arrays could be caused. CVE-2018-0933 and CVE-2018-0934 were assigned to those bugs. Note that all the patches revised the `BoxStackInstance` function in the `JavascriptArray.cpp` file.

Among the 77 patched files, 26 (33.8%) files are patched at least twice due to the reported CVEs. These examples demonstrate that JS engine vulnerabilities often arise from files that were patched for other bugs. Considering that these patches are often checked with regression tests, mutating an existing JS test may trigger a new vulnerability whose root cause lies in the patched files that this test already covered.

Observation 1. JS engine vulnerabilities often arise from the same file patched for different bugs.

We also measured the syntactic similarity between JS code from the PoC exploits and 2,038 JS files obtained from regression test suites maintained by ChakraCore. Note that a regression test suite consists of JS tests that trigger previously patched bugs and check expected outcomes with adversarial test input. In particular, we gathered the regression test files from the ChakraCore version released in August 2016, which is one month ahead of the patching date of the earliest vulnerability. Therefore, the regression test files were not affected by any of the studied vulnerabilities.

```
1 var v0 = {};
2 for (var v1 = 0; v1 < 5; v1++) {
3     v0[v1] = v1 + 5;
4 }
```

Figure 1: Example of a normalized JS file.

To measure the similarity, we normalized the identifiers in the regression test files as well as the PoC exploits. Specifically, we renamed each identifier for variables and functions to have a sequential number and a common prefix as their name. We then parsed the normalized JS files down to ASTs.

We extracted a set of unit subtrees with a depth of one

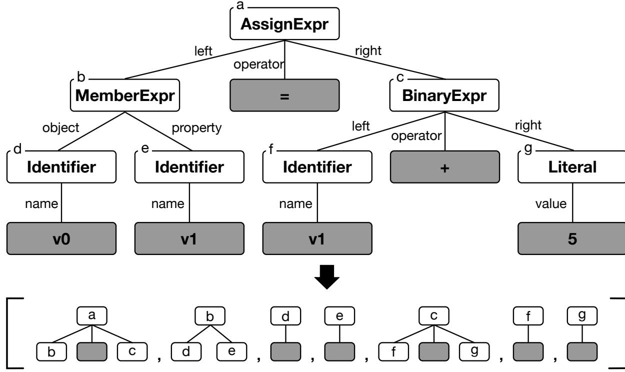


Figure 2: Fragmentizing an AST from the example in Figure 1.

from each AST. For a given AST, we extracted a unit subtree from each internal node. Thus, the number of extracted unit subtrees becomes the number of AST internal nodes. We call such a unit subtree a *fragment*, as formally defined in §5. Note that the root node of each fragment is an internal node of the AST. It also corresponds to a leaf node in another fragment, except the fragment with the root node of the original AST.

Figure 2 illustrates the fragmentation results for a JS file listed in Figure 1. The upper side of the figure shows an AST subtree obtained from the Esprima JS parser [21]. This subtree corresponds to Line 3. The bottom of the figure presents fragments from this subtree.

We also divided each PoC that triggers a CVE into fragments and then counted how many fragments existed in the regression test suites. Figure 3 depicts the number of PoC files whose common fragment percentage is over each percentage threshold. We found that all the fragments (100%) from 10 PoC exploits already existed in the regression test files. More than 96% of the fragments in the 42 PoC exploits and 90% of the fragments in the 63 PoC exploits existed in the regression test as well. On average, 95.9% of the fragments from the PoC exploits were found in the regression test files.

Observation 2. More than 95% of the fragments syntactically overlap between the regression tests and the PoC exploits.

Both observations imply that it is likely to trigger a new security vulnerability by assembling code fragments from existing regression test suites, which is the primary motivation for this study, as we describe in §4.

4 Overview

We present Montage, an NNLM-driven fuzzer, which automatically finds bugs in JS engines. Recall that the overall design of Montage is driven by two observations: (1) security bugs often arise from files that were previously patched for different causes, and (2) the JS test code that triggers security-related bugs heavily reuses AST fragments found in the existing regression test sets.

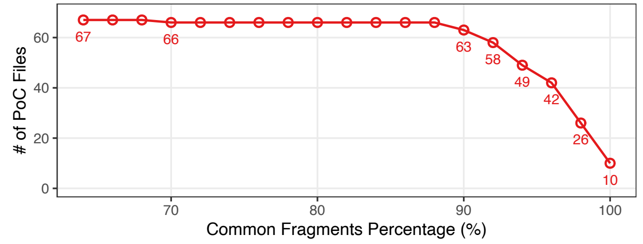


Figure 3: The number of all PoC files whose common fragment percentages are greater than varying percentages.

We propose a novel fuzzing technique that captures these observations. We train an NNLM to capture the syntactic and semantic relationships among fragments from the regression test sets. When generating a new JS test, Montage mutates the AST of a given JS regression test. It replaces a subtree of the AST with a new subtree, using the trained NNLM. Thus, each generated test stems from a given regression test that checks previously patched or buggy logic, thereby, capturing the first observation. At the same time, it invokes functionalities in different execution contexts by assembling existing fragments under the guidance of the NNLM, which addresses the second observation.

Figure 4 shows the overall workflow of Montage. Phase I prepares the training instances from given regression test suites. Each training instance is a sequence of AST unit subtrees, called *fragments*. Phase II trains an NNLM that learns compositional relationships among fragments. These two phases are one-time setup procedures. Phase III generates JS tests by leveraging the trained model.

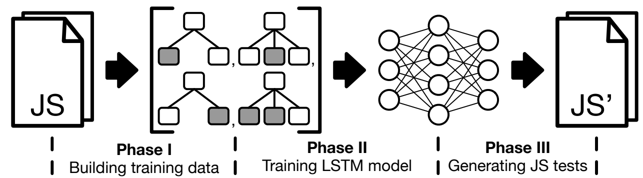


Figure 4: Overview of Montage.

Phase I begins with a given training set of JS regression test files. It parses each JS file into an AST and normalizes identifiers that appeared in the AST to deduplicate function and variable names. Figure 1 shows a normalized JS file example. Each appeared variable name is changed into a common name, such as `v0` or `v1`. From a normalized AST tree, Phase I then extracts multiple unit subtrees, each of which is called a *fragment*. For each node in the AST, Montage recursively slices a unit subtree of depth one. Each of the sliced subtrees becomes a fragment of the AST. It then emits the sequence of these fragments, produced by the pre-order traversal of their root nodes in the normalized AST tree.

Phase II trains the NNLM given a set of fragment sequences. From a given fragment sequence of an arbitrary length, we design the NNLM to suggest the next fragments, which are likely to appear after this fragment sequence. This framing is a key contribution of this paper. Note that it is not straightforward to model the inherent structural relationships of an AST in such a way that a language model can learn. By leveraging the fragments encapsulating the structural relationships of ASTs, we encode a given AST into fragment sequences. Considering that a vast volume of natural language NNLMs have been trained upon word sequences, this fragment sequencing eases the application of existing prevailing NNLMs for generating JS tests.

Here, the objective is to train the NNLM to learn compositional relationships among fragments so that the JS test code generated from the trained model reflects the syntax and semantics of the given training set, which is the regression testing set of JS engines.

Phase III generates a new JS test by leveraging the trained model and the AST of a regression test. Given a set of ASTs from regression test suites, it randomly picks a seed AST. Then, it randomly selects a subtree for Montage to replace. When generating a new subtree, Montage considers a *context*, the sequence of all fragments that precedes the selected subtree. Montage iteratively appends fragments from the root node of the selected subtree while considering its context.

Because the current AST is assembled from fragments, it is expected that some variables and function identifiers in the AST nodes are used without proper declarations. Montage, thus, resolves possible reference errors by renaming them with the declared identifiers. Finally, Montage checks the generated test and reports a bug if the code crashes the target JS engine.

Other model guided approaches. Previous studies presented language models, which can predict the lexical code tokens in source code. Such framing of language models has been vastly studied while addressing code completion problems [40, 53]. However, the generation of an executable test is more challenging than the code completion problem that predicts a limited number of semantically correct lexical tokens. To our knowledge, the PDF fuzzer proposed by Singh *et al.* [16] is the first system that employs a character-level RNN model to generate PDF tests. We evaluated whether our fragment-based approach performs better than the character-level RNN model approach in finding JS engine bugs (see §7.5).

5 Design

The design goal of Montage is to generate JS test inputs that can trigger security vulnerabilities in JS engines, which (1) reflect the syntactic and semantic patterns of a given JS training set, and (2) trigger no reference errors.

It is a technical challenge to frame the problem of teaching a language model the semantic and syntactic patterns of training code. We address this challenge by abstracting the hierarchical structure by AST subtrees, which we refer to as fragments. We then enable the language model to learn the compositional relationships between fragments.

We propose a novel code generation algorithm that leverages a trained language model. We harness an existing JS code that is already designed to trigger JS engine defects. Montage alters this existing JS code by replacing one of its AST subtrees with a new subtree that the trained language model generates. Thus, Montage is capable of generating a new JS test, semantically similar to the regression test case that triggers a previously reported bug. We expect that this new JS test triggers a new bug in a different execution context.

5.1 Phase I: Building Training Data of Fragment Sequences

Phase I prepares training instances using a given training set. It conducts *parsing* and *fragmentation*.

5.1.1 Parsing and Normalizing

Phase I builds an AST by parsing each JS file in a training set and normalizes the parsed AST. Because the training set includes a variety of JS files from various developers, identifier naming practices are not necessarily consistent. Thus, it is natural that the training files have diverse variable and function names across different JS files. Consider two JS files that contain a JS statement `var b = a + 1` and `var c = d + 1`, respectively. Both have the same AST structure and semantics, but different identifiers.

This pattern increases the size of unnecessary vocabulary for a language model to learn, rendering the model evaluation expensive as it requires more training instances. To have concise ASTs with consistent identifier names, we rename all the variable and function identifiers in the ASTs.

Specifically, for each declared variable identifier, we assign a sequential number in the order of their appearance in a given AST. We then replace each variable name with a new name that combines a common prefix and its sequential number, such as `v0` and `v1`. We also apply the same procedure to function identifiers, e.g., `f0` and `f1`. We deliberately exclude language-specific built-in functions and engine objects from the normalization step as normalizing them affects the semantics of the original AST. For an `eval` function that dynamically evaluates a given string as the JS code, we first extract the argument string of the `eval` function and strip it out as the JS code when the argument is a constant string. Subsequently, we normalize identifiers in the JS code stripped out from the `eval` argument.

As our training set is derived from regression tests of JS engines, JS files in the set make heavy use of predefined

functions for testing purposes. Therefore, we manually identified such vendor-provided testing functions and ignore them during the normalization step. That is, we treated common testing functions provided by each JS engine vendor as a built-in function and excluded them from normalization.

5.1.2 Fragmentation

Montage slices each normalized AST into a set of subtrees while ensuring that the depth of each subtree is one. We call such a unit subtree as a *fragment*.

We represent an AST T with a triple (N, E, n_0), where N is the set of nodes in T, E is the set of edges in T, and n_0 is the root node of T. We denote the immediate children of a given AST node n_i by $C(n_i)$, where n_i is a node in N. Then, we define a subtree of T where the root node of the subtree is n_i. When there is such a subtree with a depth of one, we call it a *fragment*. We now formally define it as follows.

Definition 1 (Fragment)**.** A fragment of $T = (N, E, n_0)$ is a subtree $T_i = (N_i, E_i, n_i)$, where

- $n_i \in N$ s.t. $C(n_i) \neq \emptyset$.
- $N_i = \{n_i\} \bigcup C(n_i)$.
- $E_i = \{(n_i, n') \mid n' = C(n_i)\}$.

Intuitively, a fragment whose root node is n_i contains its children and their tree edges. Note that each fragment inherently captures an exercised production rule of the JS language grammar employed to parse the AST. We also define the *type* of a fragment as the non-terminal symbol of its root node n_i. For instance, the first fragment at the bottom side of Figure 2 corresponds to the assignment expression statement in Line 3 of Figure 1. The fragment possesses four nodes whose root node is the non-terminal symbol of an `AssignmentExpression`, which becomes the *type* of this fragment.

Montage then generates a sequence of fragments by performing the pre-order traversal on the AST. When visiting each node in the AST, it emits the fragment whose root is the visited node. The purpose of the pre-order sequencing is to sort fragments by the order of their appearance in the original AST. For example, the bottom side of Figure 2 shows the sequence of seven fragments obtained from the AST subtree in the figure.

We model the compositional relationships between fragments as a pre-order sequencing of fragments so that an NNLM can predict the next fragment to use based on the fragments appearing syntactically ahead. In summary, Phase I outputs the list of fragment sequences from the training set of normalized ASTs.

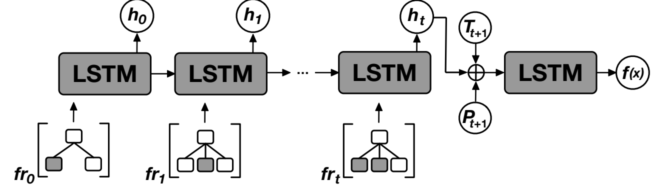


Figure 5: Architecture of Montage LSTM model. $\oplus$ in the figure denotes a concatenation.

5.2 Phase II: Training an LSTM Model

All distinct fragments become our vocabulary for the NNLM to be trained. Before training, we label the fragments whose frequency is less than five in the training set as out-of-vocabulary (OoV). This is a standard procedure for building a language model to prune insignificant words [22, 40, 43].

Each fragment sequence represents a JS file in the training set. This sequence becomes a training instance. We build a statistical language model from training instances so that the model can predict the next fragment based on all of its preceding fragments, which is considered as a *context*. This way, the model considers each fragment as a lexicon, and thereby, suggests the next probable fragments based on the current context.

Training objectives. The overall objective is to model a function $f : X \to Y$ such that $y \in Y$ is a probability distribution for the next fragment fr_{t+1}, given a fragment sequence $x = [fr_0, fr_1, ..., fr_t] \in X$, where fr_i denotes each fragment at time step i. Given x, the model is trained to (1) predict the correct next fragment with the largest probability output and (2) prioritize fragments that share the same *type* with the true fragment over other types of fragments. Note that this training objective accords with our code generation algorithm in that Montage randomly selects the next fragment of a given type from the top k suggestions (see §5.3).

LSTM. To implement such a statistical language model, we take advantage of the LSTM model [23]. Figure 5 depicts the architecture of Montage LSTM model. Our model consists of one projection, one LSTM, and one output layers. The projection layer is an embedding layer for the vocabulary where each fragment has a dimension size of 32. When fr_t is passed into the model, it is converted into a vector, called *embedding*, after passing the projection layer.

Then, the *embedding* vector becomes one of the inputs for the LSTM layer with a hidden state size of 32. At each time step, the LSTM cell takes three inputs: (1) a hidden state h_{t-1} and (2) a cell state c_{t-1} from the previous time step; and (3) the embedding of a new input fragment. This architecture enables the model to predict the next fragment based on the cumulative history of preceding fragments. In other words, the LSTM model is not limited to considering a fixed number of preceding fragments, which is an advantage of using an RNN model.

The output of the LSTM layer h_t is then concatenated with

two other vectors: (1) the type embedding T_{t+1} of the next fragment, and (2) the fragment embedding P_{t+1} of the parent fragment of the next fragment in its AST. The concatenated vector is now fed into the final output layer and it outputs a vector $f(x)$ of vocabulary size to predict the next fragment.

Loss function. To address our training objectives, we defined a new loss function that rewards the model to locate type-relevant fragments in its top suggestions. The LSTM model is trained to minimize the following empirical loss over the training set $(x,y) \in D$.

$$
\begin{aligned}
g(x) &= \mathrm{softmax}(f(x)) \\
L_D(f) &= \frac{1}{|D|} \sum_{(x,y)\in D} l_1(g(x),y) + l_2(g(x),y)
\end{aligned}
\tag{1}
$$

As shown in Equation 1, the loss function has two terms: l_1 and l_2. Note that these terms are designed to achieve our two training objectives, respectively.

$$
\begin{aligned}
l_1(g(x),y) &= -\sum_{i=1}^{N} y_i \log g(x)_i \\
l_2(g(x),y) &= \sum_{i\in \mathrm{top}(n)} g(x)_i - \sum_{j\in \mathrm{type}(y)} g(x)_j,
\end{aligned}
\tag{2}
$$

Equation 2 describes each term in detail. In the equation, n denotes the number of fragments whose types are same as that of the true fragment. $\mathrm{top}(n)$ and $\mathrm{type}(y)$ indicate functions that return the indices of top n fragments and fragments of the true type, respectively.

l_1 is a cross entropy loss function, which has been used for common natural language models [29,34]. l_2 is employed for rewarding the model to prioritize fragments that have the same type as the true fragment. We formally define l_2 as a **type error**. It is a gap between two values: the sum of the model output probabilities corresponding to (1) top n fragments and (2) fragments of the true type.

By reducing the sum of l_1 and l_2 while training, the model achieves our training objectives. Intuitively, the LSTM model is trained not only to predict the correct fragment for a given context, but also to locate fragments whose types are same as the correct fragment in its top suggestions.

The fundamental difference of Montage from previous approaches that use probabilistic language models [41,56] lies in the use of fragments. To generate JS code, TreeFuzz [41] and SkyFire [56] use a PCFG and PCSG to choose the next AST production rule from a given AST node, respectively. SkyFire defines its context to be sibling and parent nodes from a given AST. It picks an AST production rule that is less frequent in the training set. In contrast, Montage selects a fragment based on the list of fragments, not AST nodes. Therefore, Montage is capable of capturing the global composition relationships among code fragments to select the next code fragment. Furthermore, Montage preserves the semantics in the training set by slicing the AST nodes into fragments, which is used as a lexicon for generating JS code. We frame the problem of training a language model to leverage fragments and their sequences, which makes Montage compatible with prevalent statistical language models.

5.3 Phase III: Generating JS Tests

Given a set of ASTs from regression tests and the LSTM model, Phase III first mutates a randomly selected seed AST by leveraging the LSTM model. Then, it resolves reference errors in the skeleton AST.

Algorithm 1 describes our code generation algorithm. The `MutateAST` function takes two configurable parameters from users.

f_{max} — The maximum number of fragments to append. This parameter controls the maximum number of fragments that a newly composed subtree can have.

k_{top} — The number of candidate fragments. Montage randomly selects the next fragment from suggestions of the k_{top} largest probabilities at each iteration.

After several exploratory experiments, we observed that bloated ASTs are more likely to have syntactical and semantic errors. We also observed that the accuracy of the model decreases as the size of an AST increases. That is, as the size of AST increases, Montage has a higher chance of failures in generating valid JS tests. We thus capped the maximum number of fragment insertions with f_{max} and empirically chose its default value to be 100. For k_{top}, we elaborate on its role and effects in detail in §7.3.

5.3.1 Mutating a Seed AST

The `MutateAST` function takes in a set of ASTs from regression tests, the trained LSTM model, and the two parameters. It then begins by randomly selecting a seed AST from the given set (Line 2). From the seed AST, it removes one randomly selected subtree (Line 3). Note that the pruned AST becomes a base for the new JS test. Finally, it composes a new subtree by leveraging the LSTM model (Lines 4-13) and returns the newly composed AST.

After selecting a seed AST in Line 2, we randomly prune one subtree from the AST by invoking the `RemoveSubtree` function. The function returns a pruned AST and the initial *context* for the LSTM model, which is a fragment sequence up to the fragment where Montage should start to generate a new subtree. This step makes a room to compose new code.

In the `while` loop in Lines 4-13, the `MutateAST` function now iteratively appends fragments to the AST at most f_{max} times by leveraging the LSTM model. The loop starts by selecting the next fragment via the `PickNextFrag` function in Line 6. The `PickNextFrag` function first queries the LSTM model to retrieve the k_{top} suggestions. From the suggestions,

Algorithm 1: Mutating a seed AST

```
Input  : A set of ASTs from regression tests (𝕋).
         The LSTM model trained on fragments (model).
         The max number of fragments to append (f_max).
         The number of candidate fragments (k_top).
Output : A newly composed AST.
1  function MutateAST(𝕋, model, f_max, k_top)
2      n_0 ← PickRandomSeed(𝕋)
3      n_0, context ← RemoveSubtree(n_0)
4      count ← 0
5      while count ≤ f_max do
6          next_frag ← PickNextFrag(model, k_top, context)
7          if next_frag = ∅ then
8              return
9          n_0 ← AppendFrag(n_0, next_frag)
10         if not IsASTBroken(n_0) then
11             break
12         context.append(next_frag)
13         count ← count + 1
14     return n_0
15 function AppendFrag(node, next_frag)
16     ℂ ← node.child()        /* Get direct child nodes. */
17     if IsNonTerminal(node) ∧ ℂ = ∅ then
18         node ← next_frag
19         return
20     for c ∈ ℂ do
21         AppendFrag(c, frag_seq)
```

the function repeats random selections until the chosen fragment indeed has a correct type required for the next fragment. If all the suggested fragments do not have the required type, the MutateAST function stops here and abandon the AST. Otherwise, it continues to append the chosen fragment by invoking the AppendFrag function.

The AppendFrag function traverses the AST in the pre-order to find where to append the fragment. Note that this process is exactly the opposite process of an AST fragmentation in §5.1.2. Because we use a consistent traversal order in Phase I and III, we can easily find whether the current node is where the next fragment should be appended. Lines 16-19 summarize how the function determines it. The function tests whether the current node is a non-terminal that does not have any children. If the condition meets, it appends the fragment to the current node and returns. If not, it iteratively invokes itself over the children of the node for the pre-order traversal.

Note that the presence of a non-terminal node with no children indicates that the fragment assembly of the AST is still in progress. The IsASTBroken function checks whether the AST still holds such nodes. If so, it keeps appending the fragments. Otherwise, the MutateAST function returns the composed skeleton AST.

We emphasize that our code generation technique based on code fragments greatly simplifies factors that a language model should learn in order to generate an AST. TreeFuzz [41] allows a model to learn fine-grained relationships among edges, nodes, and predecessors in an AST. Their approach requires to produce multiple models each of which covers a specific property that the model should learn. This, though, brings the unfortunate side-effects of managing multiple models and deciding priorities in generating an AST when the predictions from different models conflict with each other. On the other hand, our approach abstracts such relationships as fragments, which becomes building blocks for generating AST. The model only learns the compositional relationships between such blocks, which makes training and managing a language model simple.

5.3.2 Resolving Reference Errors

Phase III resolves the reference errors from the generated AST, which appear when there is a reference to an undeclared identifier. It is natural for the generated AST to have reference errors since we assembled fragments that are used in different contexts across various training files. The reference error resolution step is designed to increase the chance of triggering bugs by making a target JS engine fully exercise the semantics of a generated testing code. The previous approaches [18, 24, 54] reuse existing AST subtrees and attach them into a new AST, which naturally causes reference errors. However, they overlooked this reference error resolution step without addressing a principled solution.

We propose a systematic way of resolving reference errors, which often accompany type errors. Specifically, we take into account both (1) statically inferred JS types and (2) the scopes of declared identifiers. Montage harnesses these two factors to generate JS test cases with fewer reference errors in the run time.

There are three technical challenges that make resolving reference errors difficult. (1) In JS, variables and functions can be referenced without their preceding declarations due to hoisting [37]. Hoisting places the declarations of identifiers at the top of the current scope in its execution context; (2) It is difficult to statically infer the precise type of each variable without executing the JS code because of no-strict type checking and dynamically changing types; and (3) Each variable has its own scope so that referencing a live variable is essential to resolve reference errors.

To address these challenges, Montage prepares a scope for each AST node that corresponds to a new block body. Montage then starts traversing from these nodes and fills the scope with declared identifiers including hoistable declarations. Each declared identifier in its scope holds the `undefined` type at the beginning.

When Montage encounters an assignment expression in its traversal, it statically infers the type of its right-hand expression via its AST node type and assigns the inferred type to its left-hand variable. Montage covers the following statically inferred types: `array`, `boolean`, `function`, `null`, `number`, `object`, `regex`, `string`, `undefined`, and `unknown`. Each

scope has an identifier map whose key is a declared identifier and value is an inferred type of the declared identifier.

To resolve reference errors, Montage identifies an undeclared variable while traversing each AST node and then infers the type of this undeclared variable based on its usage. A property or member method reference of such an undeclared variable hints to Montage to infer the type of the undeclared variable. For instance, the `length` property reference of an undeclared variable assigns the `string` type to the undeclared variable. From this inferred type, Montage replaces the undeclared identifier with a declared identifier when its corresponding type in the identifier map is same as the inferred type. If the inferred type of an undeclared variable is `unknown`, it ignores the type and randomly picks one from the list of declared identifiers. For all predefined and built-in identifiers, Montage treats them as declared identifiers.

6 Implementation

We implemented Montage with 3K+ LoC in Python and JS. We used Esprima 4.0 [21] and Escodegen 1.9.1 [51] for parsing and printing JS code, respectively. As both libraries work in the Node.js environment, we implemented an inter-process pipe channel between our fuzzer in Python and the libraries.

We implemented the LSTM models with PyTorch 1.0.0 [52], using the L2 regularization technique with a parameter of 0.0001. The stochastic gradient descent with a momentum factor of 0.9 served as an optimizer.

We leveraged the Python subprocess module to execute JS engines and obtain their termination signals. We only considered JS test cases that crash with `SIGILL` and `SIGSEGV` meaningful because crashes with other termination signals are usually intended ones by developers.

To support open science and further research, we publish Montage at `https://github.com/WSP-LAB/Montage`.

7 Evaluation

We evaluated Montage in several experimental settings. The goal is to measure the efficacy of Montage in finding JS engine bugs, as well as to demonstrate the necessity of an NNLM in finding bugs. We first describe the dataset that we used and the experimental environment. Then, we demonstrate (1) how good a trained Montage NNLM is in predicting correct fragments (§7.2), (2) how we set a k_{top} parameter for efficient fuzzing (§7.3), (3) how many different bugs Montage discovers, which other existing fuzzers are unable to find (§7.4), and (4) how much the model contributes to Montage finding bugs and generating valid JS tests (§7.5). We conclude the evaluation with field tests on the latest JS engines (§7.6). We also discuss case studies of discovered bugs (§7.7).

7.1 Experimental Setup

We conducted experiments on two machines running 64-bit Ubuntu 18.04 LTS with two Intel E5-2699 v4 (2.2 GHz) CPUs (88 cores), eight GTX Titan XP DDR5X GPUs, and 512 GB of main memory.

Target JS engine. The ChakraCore GitHub repository has managed the patches for all the reported CVEs by the commit messages since 2016. That is, we can identify the patched version of ChakraCore for each known CVE and have ground truth that tells whether found crashes correspond to one of the known CVEs [9]. Therefore, we chose an old version of ChakraCore as our target JS engine. We specifically performed experiments on ChakraCore 1.4.1, which was the first stable version after January 31, 2017.

Data. Our dataset is based on the regression test sets of Test262 [13] and the four major JS engine repositories at the version of January 31, 2017: ChakraCore, JavaScriptCore, SpiderMonkey, and V8. We excluded test files that ChakraCore failed to execute because of their engine-specific syntax and built-in objects. We did not take into account files larger than 30 KB because large files considerably increase the number of unique fragments with low frequency. In total, we collected 1.7M LoC of 33,486 unique JS files.

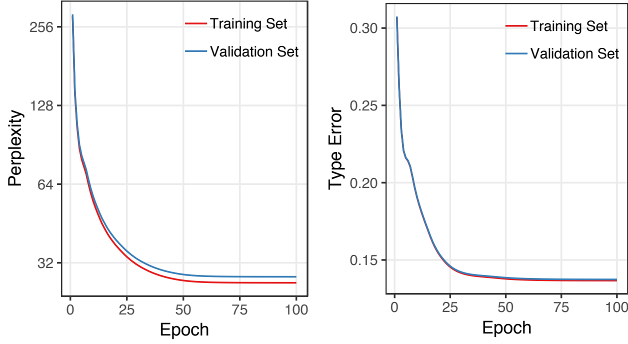
Temporal relationships. Montage only used the regression test files committed before January 31, 2017, and performed fuzz testing campaigns on the first stable version after January 31, 2017. Thus, the bugs that regression tests in the training dataset check and the bugs that Montage is able to find are disjoint. We further confirmed that all CVEs that Montage found were patched after January 31, 2017.

Fragments. From the dataset, we first fragmented ASTs to collect 134,160 unique fragments in total. On average, each training instance consisted of 118 fragments. After replacing less frequent fragments with OoVs, they were reduced to 14,518 vocabularies. Note that most replaced fragments were string literals, e.g., bug summaries, or log messages.

Bug ground truth. Once Montage found a JS test triggering a bug, we ran the test against every patched version of ChakraCore to confirm whether the found bug matches one of the reported CVEs. This methodology is well-aligned with that of Klees *et al.* [31], which suggests counting distinct bugs using ground truth. When there is no match, the uniqueness of a crash was determined by its instruction pointer address without address space layout randomization (ASLR). We chose this conservative setting to avoid overcounting the number of found bugs [36].

7.2 Evaluation of the LSTM Model

To train and evaluate the LSTM model of Montage, we performed a 10-fold cross-validation on our dataset. We first randomly selected JS files for the test set, which accounted for 10% of the entire dataset. We then randomly split the



(a) Perplexity of the model. (b) Type error proportion.

Figure 6: Perplexity and type error proportion of the LSTM model measured against the training and validation sets over epochs. They are averaged across the 10 cross-validation sets.

remaining files into 10 groups. We repeated holding out one group for the validation set and taking the rest of them for the training set for 10 times.

Figure 6 illustrates the *perplexity* and *type error* of the LSTM model measured on the training and validation sets. Recall that the loss function of the model is a sum of the *log perplexity* and *type error* (§5.2).

Perplexity. Perplexity measures how well a trained model predicts the next word that follows given words without perplexing. It is a common metric for evaluating natural language models [29, 34]. A model with a lower perplexity performs better in predicting the next probable fragment. Note from Figure 6a that the perplexities for both the training and validation sets decrease without a major difference as training goes on.

Type error. Type error presents how well our model predicts the correct type of a next fragment (recall §5.2). A model with a low type error is capable of predicting the fragments with the correct type in its top predictions. Note from Figure 6b that the type errors for both the training and validation sets continuously decrease and become almost equal as the epoch increases.

The small differences of each perplexity and type error between the training set and validation set demonstrate that our LSTM model is capable of learning the compositional relations among fragments without overfitting or underfitting.

We further observed that epoch 70 is the optimal point at which both valid perplexity and valid type errors start to plateau. We also noticed that the test perplexity and test type errors at epoch 70 are 28.07 and 0.14, respectively. Note from Figure 6 that these values are close to those from the validation set. It demonstrates that the model can accurately predict fragments from the test set as well. Thus, for the remaining evaluations, we selected the model trained up to epoch 70, which took 6.6 hours on our machine.

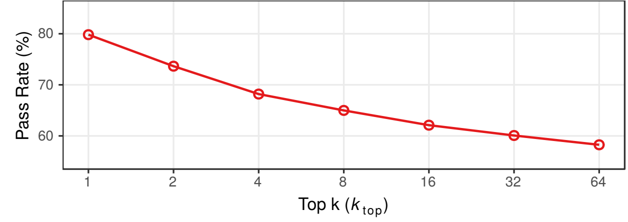


Figure 7: The pass rate of generated JS tests over k_{top}.

7.3 Effect of the k_{top} Parameter

Montage assembles model-suggested fragments when replacing an AST subtree of a given JS code. In this process, Montage randomly picks one fragment from the *Top-k* (k_{top}) suggestions for each insertion. Our key intuition is that selecting fragments from the *Top-k* rather than *Top-1* suggestion helps Montage generate diverse code, which follows the pattern of JS codes in our dataset but slightly differs from them. We evaluated the effect of the k_{top} with seven different values varying from 1 to 64 to verify our intuition.

We measured the pass rate of generated JS tests. A pass rate is a measurement unit of demonstrating how many tests a target JS engine executes without errors among generated test cases. To measure the pass rate, we first generated 100,000 JS tests with each k_{top} value. We only considered five runtime errors defined by the ECMAScript standard as errors [27]. We then ran Montage for 12 hours with each k_{top} value to count the number of crashes found in ChakraCore 1.4.1.

Figures 7 and 8 summarize our two experimental results, respectively. As shown in Figure 7, the pass rate of Montage decreases from 79.82% to 58.26% as the k_{top} increases. This fact demonstrates that the suggestion from the model considerably affects the generation of executable JS tests. It is also consistent with the experimental results from Figure 8b, in that Montage finds fewer total crashes when considering more fragment suggestions in generating JS tests. Note that Michael *et al.* [41] demonstrated that their TreeFuzz achieved a 14% pass rate, which is significantly lower than that Montage achieved.

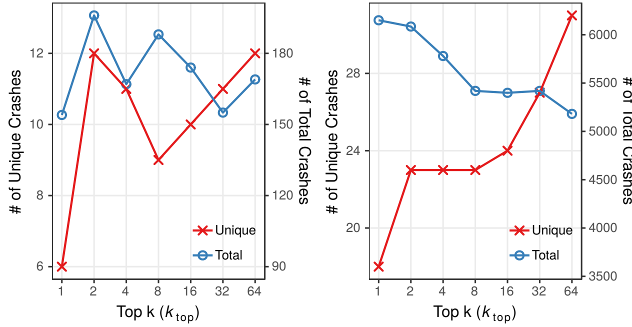
However, note from Figure 8b that the number of unique crashes increases, as k_{top} increases, unlike that of total crashes. This observation supports our intuition that increasing the k_{top} helps Montage generate diverse JS tests that trigger undesired crashes in the JS engines. Figure 8 also shows that Montage found more crashes from the debug build than the release build. Moreover, unlike the debug build, the results for the release build did not show a consistent pattern. We believe these results are mainly due to the nature of the debug build. It behaves more conservatively with inserted assertion statements, thus producing crashes for every unexpected behavior.

As Klees *et al.* [31] stated, fuzzers should be evaluated using the number of unique crashes, not that of crashing inputs. For both release and debug builds of ChakraCore 1.4.1,

Table 1: The number of bugs found with four fuzzers and four different approaches: Montage, CodeAlchemist (CA), jsfunfuzz, and IFuzzer; random selection, Markov chain, char/token-level RNN, and Montage (k_{top} = 64) without resolving reference errors. We marked results in bold when the difference between Montage and the other approach is statistically significant.

Build	Metric	# of Unique Crashes (Known CVEs)							
		Montage	CA	jsfunfuzz	IFuzzer	random	Markov	ch-RNN	Montage †
Release	Median	23 (7)	15 (4)	27 (3)	4 (1)	12 (3)	19 (6)	1 (0)	12 (4)
	Max	26 (8)	15 (4)	31 (4)	4 (2)	15 (4)	22 (7)	1 (1)	13 (5)
	Min	20 (6)	14 (3)	25 (3)	0 (0)	10 (3)	16 (5)	0 (0)	11 (4)
	Stdev	2.30 (0.84)	0.55 (0.55)	2.19 (0.45)	1.79 (0.71)	2.07 (0.45)	2.39 (0.84)	0.45 (0.55)	0.84 (0.45)
	p-value	N/A	**0.012 (0.012)**	**0.029 (0.012)**	**0.012 (0.012)**	**0.012 (0.012)**	**0.037** (0.144)	**0.012 (0.012)**	**0.012 (0.012)**
Debug	Median	49 (12)	26 (6)	27 (4)	6 (1)	31 (7)	44 (11)	3 (0)	41 (9)
	Max	52 (15)	30 (6)	29 (5)	8 (3)	34 (7)	50 (12)	4 (1)	43 (10)
	Min	45 (11)	24 (4)	24 (4)	2 (0)	27 (6)	42 (8)	1 (0)	38 (8)
	Stdev	2.70 (1.64)	2.61 (0.89)	2.12 (0.45)	2.41 (1.10)	2.88 (0.45)	3.27 (1.67)	1.10 (0.5)	1.82 (0.84)
	p-value	N/A	**0.012 (0.012)**	**0.012 (0.012)**	**0.012 (0.012)**	**0.012 (0.012)**	0.144 (0.298)	**0.012 (0.012)**	**0.012 (0.012)**
Both	Total	133 (15)	65 (7)	57 (4)	22 (3)	72 (9)	109 (14)	10 (2)	74 (10)
	Common	36 (8)	22 (2)	17 (3)	1 (0)	29 (6)	37 (8)	1 (0)	37 (7)

† Montage without resolving reference errors.



(a) Crashes on the release build. (b) Crashes on the debug build.

Figure 8: The number of total and unique crashes found in ChakraCore 1.4.1 while varying the k_{top}.

Montage found the largest number of unique crashes when the k_{top} was 64. Therefore, we picked the k_{top} to be 64 for the remaining experiments.

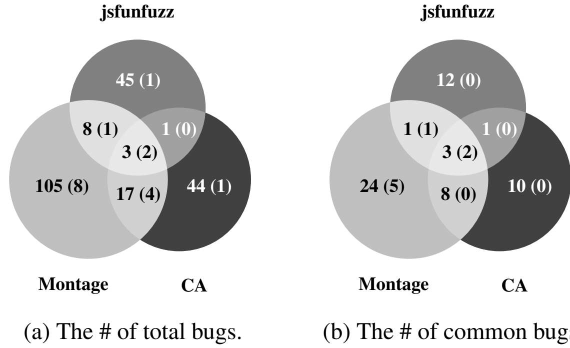
7.4 Comparison to State-of-the-art Fuzzers

To verify the ability to find bugs against open-source state-of-the-art fuzzers, we compared Montage with CodeAlchemist [20], jsfunfuzz [38], and IFuzzer [54]. jsfunfuzz and IFuzzer have been used as a controlled group in the comparison studies [20, 24]. Furthermore, CodeAlchemist, which assembles its building blocks in a semantics-aware fashion, and IFuzzer, which employs an evolutionary approach with genetic programming, have in common with Montage in that they take in a corpus of JS tests. Since Montage, CodeAlchemist, and IFuzzer start from given seed JS files, we fed them the same dataset collected from the repositories of Test262 and the four major JS engines. For fair comparison, we also configured jsfunfuzz to be the version of January 31, 2017, on which we collected our dataset (recall §7.1).

We ran all four fuzzers on ChakraCore 1.4.1 and counted the number of found unique crashes and known CVEs. Since most fuzzers depend on random factors, which results in a high variance of fuzzing results [31], we conducted five trials; each trial lasted for 6,336 CPU hours (72 hours × 88 cores). We intentionally chose such a long timeout, because fuzzers using evolutionary algorithms, such as IFuzzer, could improve their bug-finding ability as more tests are generated. Note that we expended a total of 31,680 CPU hours on the five trials of each fuzzer. Because Montage took 6.6 hours to train its language model and used this model for the five trials, we set the timeout of other fuzzers 1.3 hours (6.6 hours / 5 trials) longer than that of Montage for fair comparison.

The Montage, CA, jsfunfuzz, and IFuzzer columns of Table 1 summarize the statistical analysis of the comparison experimental results. For the release build, Montage found the largest number of CVEs, whereas jsfunfuzz still discovered more unique crashes than others. For the debug build, Montage outperformed all others in finding both unique crashes and CVEs. We performed two-tailed Mann Whitney U tests and reported p-values between Montage and the other fuzzers in the table. We verified that all results are statistically significant with p-values less than 0.05.

The last two rows of the table show the number of total and common bugs found in the five trials from the release and debug builds, respectively. We counted common bugs when Montage found these bugs in every run of the five campaigns. When a bug was found during at least one campaign, they are



(a) The # of total bugs. (b) The # of common bugs.

Figure 9: The comparison of unique crashes (known CVEs) found by Montage, CodeAlchemist (CA), and jsfunfuzz.

counted in the total bugs. Note that Montage found at least 2.14× more CVEs compared to others in a total of the five trials. We believe that these results explain the significance of Montage in finding security bugs compared to the other state-of-the-art fuzzers.

We also compared the bugs discovered by each fuzzer. Figure 9 depicts the Venn diagrams of unique bugs found in ChakraCore 1.4.1. These Venn diagrams present the total and common bugs that each fuzzer found, corresponding to the last two rows of Table 1. We excluded IFuzzer from the figure because all found CVEs were also discovered by Montage.

Note from Figure 9a that Montage identified 105 unique crashes in total, including eight CVEs that were not found by CodeAlchemist and jsfunfuzz. Furthermore, Montage discovered all CVEs that were commonly found in the five trials of CodeAlchemist and jsfunfuzz, as shown in Figure 9b. However, CodeAlchemist and jsfunfuzz also identified a total of 45 and 46 unique bugs that were not found by Montage, respectively. These results demonstrate that Montage plays a complementary role against the state-of-the-art fuzzers in finding distinctive bugs.

Performance over time. Figure 10 shows the number of CVEs that Montage found over time. The number increases rapidly in the first 1,144 CPU hours (13 hours × 88 cores) of the fuzzing campaigns; however, Montage finds additional bugs after running for 2,640 CPU hours (30 hours × 88 cores), thus becoming slow to find new vulnerabilities.

7.5 Effect of Language Models

Montage generates JS tests by assembling language model-suggested fragments. Especially, it takes advantage of the LSTM model to reflect the arbitrary length of preceding fragments when predicting the next relevant fragments. However, Montage can leverage any other prevailing language models by its design, and the language model it employs may substantially affect its fuzzing performance. Therefore, to analyze the efficacy of the LSTM model in finding bugs, we first conducted a comparison study against two other approaches: (1)

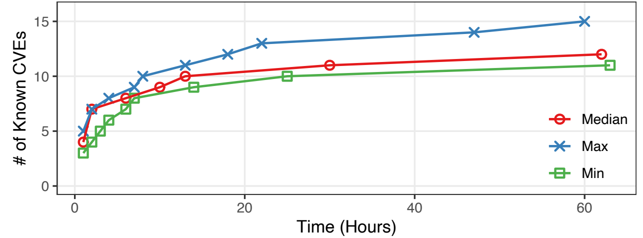


Figure 10: The number of CVEs found by Montage over time.

a random fragment selection, and (2) Markov model-driven fragment selection.

The former approach is the baseline for Montage where fragments are randomly appended instead of querying a model. The latter approach uses a Markov model that makes a prediction based on the occurrence history of the preceding two fragments. Specifically, we tailored the code from [25] to implement the Markov chain.

Additionally, we compared our approach against a character/token-level RNN language model-guided selection. It leverages an NNLM to learn the intrinsic patterns from training instances, which is in common with ours. Recently proposed approaches [11,16,32], which resort to an NNLM to generate highly structured inputs, adopted an approach more or less similar to this one.

Note that there is no publicly available character/token-level RNN model to generate JS tests. Thus, we referenced the work of Cummins *et al.* [11] to implement this approach and trained the model from scratch. To generate test cases from the trained model, we referenced the work of Liu *et al.* [32] because their approach is based on the seed mutation like our approach.

The random, Markov, and ch-RNN columns of Table 1 summarize the number of crashes found by each approach. We conducted five fuzzing campaigns, each of which lasted 72 hours; all the underlying experimental settings are identical to those in §7.4. Note that we conducted resolving reference errors and fed the same dataset as Montage when evaluating the aforementioned three models. Montage outperformed the random selection and character/token-level RNN methods in the terms of finding crashes and security bugs; thus, yielding p-values under 0.05, which suggests the superiority of Montage with statistical significance.

When comparing the metrics from release and debug build between Montage and the Markov chain approach, Montage performed better. Montage found more unique bugs in total as well. However, the Mann Whitney U test deems the difference insignificant. Nevertheless, we emphasize that Montage is capable of composing sophisticated subtrees that the Markov chain easily fails to generate. For instance, Montage generated a JS test triggering CVE-2017-8729 by appending 54 fragments, which the Markov chain failed to find. We provide more details of this case in §7.7.1.

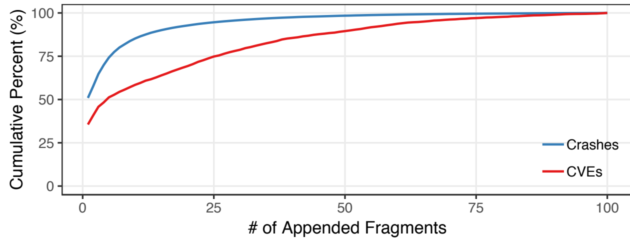


Figure 11: Empirical CDF of the number of appended fragments against JS tests causing crashes in ChakraCore.

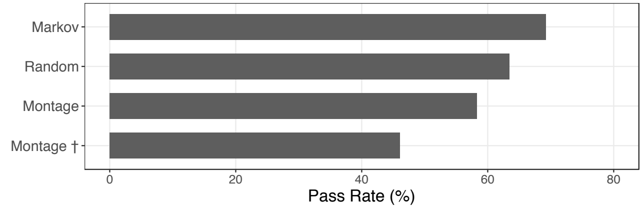


Figure 12: The pass rate measured against four different approaches: Montage (k_{top} = 64) with and without resolving reference errors, random selection, and Markov model. Montage without resolving reference errors is denoted by †.

To evaluate the effectiveness of the LSTM model, we further analyzed the number of fragments Montage appended to generate JS tests that caused ChakraCore 1.4.1 to crash in the experiment from §7.4.

Figure 11 shows the cumulative distribution function (CDF) of the number of inserted fragments against 169,072 and 5,454 JS tests causing crashes and known CVEs, respectively. For 90% of JS tests that caused the JS engine to crash, Montage only assembled fewer than 15 fragments; however, it appended up to 52 fragments to generate 90% of JS tests that found the known CVEs. This demonstrates that Montage should append more fragments suggested by the model to find security bugs rather than non-security bugs. It also denotes that the random selection approach suffers from finding security bugs. Note that Table 1 also accords with this result.

From the aforementioned studies, we conclude that the LSTM model trained on fragments is necessary for finding bugs in the JS engines.

Resolving reference errors. We evaluated the importance of the reference error resolution step (recall §5.3.2) in finding JS engine bugs. Specifically, we ran Montage with the same settings as other approaches while letting it skip the reference error resolution step but still leverage the same LSTM model. The last column of Table 1 demonstrates that Montage finds fewer bugs if the resolving step is not applied, denoting that the error resolving step improves the bug-finding capability of Montage. However, Montage still found more bugs than the other state-of-the-art fuzzers and the random approach even without the resolving step. Considering the random approach also takes advantages of the error resolution step, the LSTM model significantly contributes to finding JS engine bugs.

Pass rate. One of the key objectives of Montage is to generate a valid JS test so that it can trigger deep bugs in JS engines. Thus, we further measured how the use of language models affects the pass rate of generated codes. A pass rate indicates whether generated test cases are indeed executed after passing both syntax and semantic checking.

Figure 12 illustrates the pass rate of 100,000 JS tests generated by the four different approaches: Montage with and without resolving reference errors, the random selection, and the Markov model. We excluded the character/token-level RNN approach because only 0.58% of the generated tests were executed without errors. Such a low pass rate could be one possible reason why this approach failed to find many bugs, as shown in Table 1. As Liu *et al.* [32] also stated in their paper, we believe this result is attributed to the lack of training instances and the unique characteristics inherent in the regression test suite.

Note from the figure that resolving reference errors increases the pass rate by 12.2%. As a result, this helped Montage to find more bugs, as shown in Table 1. On the other hand, the pass rates of the random selection and Markov model-guided approach were 5.2% and 11% greater than that of Montage, respectively. We manually inspected the JS tests generated by the random selection and the Markov model-guided approaches. We concluded that these differences stem from appending a small number of fragments. For instance, if a model always replaces one fragment, such as a string literal, from the seed file, all generated JS tests will be executed without errors.

7.6 Field Tests

We evaluated the capability of Montage in finding real-world bugs. *We have run Montage for 1.5 months on the latest production versions of the four major engines:* ChakraCore, JavaScriptCore, SpiderMonkey, and V8. For this evaluation, we collected datasets from the repository of each JS engine at the version of February 3, 2019. We additionally collected 165 PoCs that triggered known CVEs as our dataset. Then, we trained the LSTM model for each JS engine.

Montage has found 37 unique bugs from the four major JS engines so far. Among the found bugs, 34 bugs were from ChakraCore. The remaining two and one bugs were from JavaScriptCore and V8, respectively. We manually triaged each bug and reported all the found bugs to the vendors. In total, *26 of the reported bugs have been patched so far*.

Especially, we reported three of the found bugs as security-related because they caused memory corruptions of the target JS engines. The three security bugs were discovered in ChakraCore 1.11.7, ChakraCore 1.12.0 (beta), and JavaScriptCore 2.23.3, respectively. Note that all of them got CVE IDs: CVE-2019-0860, CVE-2019-0923, and CVE-2019-8594. Particularly, we were rewarded for the bugs found in ChakraCore

with a bounty of 5,000 USD.

Our results demonstrate that Montage is capable of finding 37 real-world JS engine bugs, including three security bugs. We further describe one of the real-world security bugs that Montage found in §7.7.3.

7.7 Case Study

To show how Montage leverages the existing structure of the regression test, we introduce three bugs that Montage found. We show two bugs that Montage found in ChakraCore 1.4.1 from the experiment in §7.4. We then describe one of the real-world security bugs found in the latest version of ChakraCore, which is already patched. Note that we minimized all test cases for ease of explanation.

7.7.1 CVE-2017-8729

```
1  (function () {
2      for (var v0 in [{
3          v1 = class {},
4          v2 = 2010
5      }.v2 = 20]) {
6          for([] in {
7              value: function() {},
8              writable: false
9          }){}
10     }
11 })();
```

Figure 13: A test code that triggers CVE-2017-8729 on ChakraCore 1.4.1.

Figure 13 shows the minimized version of a generated test that triggers CVE-2017-8729 on ChakraCore 1.4.1. From its seed file, Montage removed the body of `FunctionExpression` and composed a new subtree corresponding to Lines 2-10. Particularly, Montage appended 54 fragments to generate the new test.

ChakraCore is supposed to reject the generated test before its execution because it has a syntax error in the `ObjectPattern` corresponding to Lines 2-5. However, assuming the `ObjectPattern` to be an `ObjectExpression`, ChakraCore successfully parses the test and incorrectly infers the type of the property `v2` to be a setter. Thus, the engine crashes with a segmentation fault when it calls the setter in Line 5. Interestingly, the latest version of ChakraCore still executes this syntax-broken JS test without errors but does not crash.

The original regression test checked the functionalities regarding a complicated `ObjectPattern`. Similarly, the generated test triggered a type confusion vulnerability while parsing the new `ObjectPattern`. Therefore, we believe that this case captures the design goal of Montage, which leverages an existing regression test and puts it in a different execution context to find a potential bug.

7.7.2 CVE-2017-8656

```
1  var v1 = {
2      'a': function () {}
3  }
4  var v2 = 'a';
5  (function () {
6      try {
7      } catch ([v0 = (v1[v2].__proto__(1, 'b'))]) {
8          var v0 = 4;
9      }
10     v0++;
11 })();
```

Figure 14: A test code that triggers CVE-2017-8656 on ChakraCore 1.4.1.

Figure 14 shows a test case generated by Montage that triggers CVE-2017-8656 on ChakraCore 1.4.1. Its seed file had a different `AssignmentExpression` as the parameter of a `CatchClause` in Line 7. From the seed AST, Montage removed a subtree corresponding to the `AssignmentExpression` in Line 7 and mutated it by appending eight fragments that the LSTM model suggested.

In the generated code, the variable `v0` is first declared as the parameter of the `CatchClause` (Line 7) and then redeclared in its body (Line 8). At this point, the ChakraCore bytecode generator becomes confused with the scope of these two variables and incorrectly selects which one to initialize. Consequently, the variable `v0` in Line 8 remains uninitialized. As the JS engine accesses the uninitialized symbol in Line 10, it crashes with a segmentation fault.

We note that the seed JS test aimed to check possible scope confusions, and the generated code also elicits a new vulnerability while testing a functionality similar to the one its seed JS test checks. Hence, this test case fits the design objective of Montage.

7.7.3 CVE-2019-0860

Figure 15 describes a JS test triggering CVE-2019-0860 on ChakraCore 1.12.0 (beta), which we reported to the vendor. Its seed file had a `CallExpression` instead of the statements in Lines 3-4. From the seed JS test, Montage removed a subtree corresponding to the `BlockStatement` of the function `f0` and appended 19 fragments to compose a new block of statements (Lines 2-4). Notably, Montage revived the `AssignmentExpression` statement in Line 2, which is a required precondition to execute the two subsequent statements and trigger the security bug.

The seed regression test was designed to test whether JS engines correctly handle referencing the `arguments` property

```
function f0(f, p = {}) {
    f.__proto__ = p;
    f.arguments = 44;
    f.arguments === 44;
}

let v1 = new Proxy({}, {});
for (let v0 = 0; v0 < 1000; ++v0) {
    f0(function () {'use strict';}, v1);
    f0(class C {}, v1);
}
```

Figure 15: A test code that triggers CVE-2019-0860 on ChakraCore 1.12.0 (beta).

of a function in the strict mode. For usual cases, JS engines do not allow such referencing; however, to place the execution context in an exceptional case, the seed JS test enables the access by adding a `Proxy` object to the prototype chain of the function `f` (Line 2). As a result, this new test is able to access and modify the property value without raising a type error (Line 3).

While performing the JIT optimization process initiated by the `for` loop in Lines 8-11, ChakraCore misses a postprocessing step of the property in Line 3, thus enabling to write an arbitrary value on the memory. Consequently, the engine crashes with a segmentation fault as this property is accessed in Line 4.

Note that the generated test case triggers a new vulnerability while vetting the same functionality that its seed tests. Moreover, the `GlobOpt.cpp` file, which is the most frequently patched file to fix CVEs assigned to ChakraCore, was patched to fix this vulnerability. Therefore, this JS test demonstrates that Montage successfully discovers bugs that it aims to find.

8 Related Work

Fuzzing. There have been a vast volume of research on generation-based fuzzing. Highly-structured file fuzzing [15, 42], protocol fuzzing [46, 47], kernel fuzzing [19, 28, 55], and interpreter fuzzing [2, 6, 12, 38, 41, 56] are representative research examples.

IMF infers the model of sequential kernel API calls to fuzz macOS kernels [19]. Dewey *et al.* [12] generated code with specified combinations of syntactic features and semantic behaviors by constraint logic programming.

Godefroid *et al.* [16] trained a language model from a large number of PDF files and let the model learn the relations between objects constituting the PDF files. Their approach of using a language model in generating tests is similar to ours *per se*, but their approach is not directly applicable to generating JS tests, which demands modeling complicated control and data dependencies.

Cummins *et al.* [11] also proposed a similar approach. They trained an LSTM language model from a large corpus of OpenCL code. Unlike Montage, they trained the model at a character/token-level, which does not consider the compositional relations among the AST subtrees.

TreeFuzz [41] is another model-based fuzzer. Its model is built on the frequencies of co-occurring nodes and edges from given AST examples. Their modeling of generating tests is not directly applicable to the prevalent state-of-the-art language models, tailored to train word sequences, not node and edge relations in ASTs.

Aschermann *et al.* [2] and Blazytko *et al.* [6] recently proposed NAUTILUS and GRIMOIRE, respectively. Both fuzzers test programs that take highly structured inputs by leveraging code coverage. Based on a given grammar, NAUTILUS generates a new JS test and checks whether it hits new code coverage for further mutation chances. Contrary to NAUTILUS, GRIMOIRE requires no user-provided components, such as grammar specification and language models, but synthesizes inputs that trigger new code coverage. As they stated, GRIMOIRE has difficulties in generating inputs with complex structures, requiring semantic information.

Previous studies of mutational fuzzing [7, 18, 24, 32, 44, 54, 57, 58] focus on altering given seeds to leverage functionalities that the seeds already test.

LangFuzz [24] is a mutational fuzzing tool that substitutes a non-terminal node in a given AST with code fragments. It iteratively replaces non-terminal nodes in the step of inserting fragments. However, LangFuzz does not consider any context regarding picking a promising candidate to cause a target JS engine crash. On the other hand, Montage is capable of learning implicit relations between fragments that may be inherent in given examples.

Liu *et al.* [32] proposed a mutation-based approach to fuzz the target program. Given a large corpus of C code, they trained a sequence-to-sequence model to capture the inherent pattern of input at character-level. Then, they leveraged the trained model to mutate the seed. Their approach suffers from the limitation that the model generates many malformed tests, such as unbalanced parenthesis.

Language model for code. Hindle *et al.* [22] measured the naturalness of software by computing the cross-entropy values over lexical code tokens in large JAVA and C applications. They also first demonstrated even count-based n-gram language models are applicable to code completion. To make more accurate suggestions for code completion, SLAMC [40] incorporated semantic information, including type, scope, and role for each lexical token. SLANG [43] lets a model learn API call sequences from Android applications. It then uses such a model to improve the precision of code completion. GraLan learns the relations between API calls from the graph of API call sequences, and ASTLan uses GraLan to fill holes in the AST to complete the code [39].

Maddison *et al.* [33] studied the generative models of natural source code based on PCFGs and source code-specific structures. Bielik *et al.* [5] suggested a new generative prob-

abilistic model of code called a probabilistic higher order grammar, which generalizes PCFGs and parameterizes the production rules on a context.

The objective of using a language model in all of the above works is to make better suggestions for code completion. However, Montage focuses on generating JS tests that should be accepted by a target JS engine.

9 Conclusion

We present Montage, the first fuzzing tool that leverages an NNLM in generating JS tests. We propose a novel algorithm of modeling the hierarchical structures of a JS test case and the relationships among such structures into a sequence of fragments. The encoding of an AST into a fragment sequence enables Montage to learn the relationships among the fragments by using an LSTM model. Montage found 37 real-world bugs in the latest JS engines, which demonstrates its effectiveness in finding JS engine bugs.

Acknowledgements

We thank anonymous reviewers for their helpful feedback and Yale Song who helped develop some of the ideas used in this paper. We are also grateful to Jihoon Kim for kindly sharing his findings, which indeed inspired our project. Finally, we thank Sunnyeo Park for collecting JS seeds used for the evaluation. This work was partly supported by (1) Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT), No.2018-0-00254, and (2) LIG Nex1.

References

[1] Ebru Arisoy, Tara N. Sainath, Brian Kingsbury, and Bhuvana Ramabhadran. Deep neural network language models. In *Proceedings of the NAACL-HLT 2012 Workshop*, pages 20–28, 2012.

[2] Cornelius Aschermann, Patrick Jauernig, Tommaso Frassetto, Ahmad-Reza Sadeghi, Thorsten Holz, and Daniel Teuchert. NAUTILUS: Fishing for deep bugs with grammars. In *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[3] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *The Journal of Machine Learning Research*, 3(1):1137–1155, 2003.

[4] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *Transactions on Neural Networks*, 5(2):157–166, 1994.

[5] Pavol Bielik, Veselin Raychev, and Martin Vechev. PHOG: Probabilistic model for code. In *Proceedings of the International Conference on Machine Learning*, pages 2933–2942, 2016.

[6] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing structure while fuzzing. In *Proceedings of the USENIX Security Symposium*, pages 1985–2002, 2019.

[7] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 725–741, 2015.

[8] Stanley F. Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics*, pages 310–318, 1996.

[9] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 197–208, 2013.

[10] Microsoft Corporation. Microsoft ChakraCore. https://github.com/Microsoft/ChakraCore.

[11] Chris Cummins and Alastair Murray. Compiler fuzzing through deep learning. In *Proceedings of the ACM International Symposium on Software Testing and Analysis*, pages 95–105, 2018.

[12] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Language fuzzing using constraint logic programming. In *Proceedings of the International Conference on Automated Software Engineering*, pages 725–730, 2014.

[13] Technical Committee 39 ECMA International. Test262. https://github.com/tc39/test262.

[14] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with LSTM. *Neural Computation*, 12:2451–2471, 1999.

[15] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 206–215, 2008.

[16] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&Fuzz: Machine learning for input fuzzing. In *Proceedings of the International Conference on Automated Software Engineering*, pages 50–59, 2017.

[17] Joshua T. Goodman. A bit of progress in language modeling. *Computer Speech & Language*, 15(4):403–434, 2001.

[18] Tao Guo, Puhan Zhang, Xin Wang, and Qiang Wei. GramFuzz: Fuzzing testing of web browsers based on grammar analysis and structural mutation. In *Proceedings of the International Conference on Informatics Applications*, pages 212–215, 2013.

[19] HyungSeok Han and Sang Kil Cha. IMF: Inferred model-based fuzzer. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 2345–2358, 2017.

[20] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines. In *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[21] Ariya Hidayat. ECMAScript parsing infrastructure for multipurpose analysis. https://www.esprima.org.

[22] Abram Hindle, Earl T. Barr, Zhendon Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the International Conference on Software Engineering*, pages 837–847, 2012.

[23] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[24] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proceedings of the USENIX Security Symposium*, pages 445–458, 2012.

[25] BuzzFeed Inc. Markovify. https://github.com/jsvine/markovify.

[26] Theori Inc. pwn.js. https://github.com/theori-io/pwnjs, 2017.

[27] ECMA International. ECMAScript language specification. https://www.ecma-international.org/ecma-262/.

[28] Dave Jones. Trinity. https://github.com/kernelslacker/trinity.

[29] Rafal Józefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *CoRR*, abs/1602.02410, 2016.

[30] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M. Rush. Character-aware neural language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 2741–2749, 2016.

[31] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[32] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. DeepFuzz: Automatic generation of syntax valid c programs for fuzz testing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1044–1051, 2019.

[33] Chris J. Maddison and Daniel Tarlow. Structured generative models of natural source code. In *Proceedings of the International Conference on Machine Learning*, pages 649–657, 2016.

[34] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Proceedings of the 11th Annual Conference of the International Speech Communication Association*, pages 1045–1048, 2010.

[35] Matt Molinyawe, Abdul-Aziz Hariri, and Jasiel Spelman. $hell on Earth: From browser to system compromise. In *Proceedings of the Black Hat USA*, 2016.

[36] David Molnar, Xue Cong Li, and David A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the USENIX Security Symposium*, pages 67–82, 2009.

[37] Mozilla. Hoisting. https://developer.mozilla.org/en-US/docs/Glossary/Hoisting.

[38] MozillaSecurity. funfuzz. https://github.com/MozillaSecurity/funfuzz.

[39] Anh Tuan Nguyen and Tien N. Nguyen. Graph-based statistical language model for code. In *Proceedings of the International Conference on Software Engineering*, pages 858–868, 2015.

[40] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. A statistical semantic language model for source code. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 532–542, 2013.

[41] Jibesh Patra and Michael Pradel. Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data. Technical Report TUD-CS-2016-14664, TU Darmstadt, 2016.

[42] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Model-based whitebox fuzzing for program binaries. In *Proceedings of the International Conference on Automated Software Engineering*, pages 543–553, 2016.

[43] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 419–428, 2014.

[44] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *Proceedings of the USENIX Security Symposium*, pages 861–875, 2014.

[45] Jesse Ruderman. Releasing jsfunfuzz and domfuzz. http://www.squarefree.com/2015/07/28/releasing-jsfunfuzz-and-domfuzz/, 2007.

[46] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *Proceedings of the USENIX Security Symposium*, pages 193–206, 2015.

[47] Juraj Somorovsky. Systematic fuzzing and testing of TLS libraries. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1492–1504, 2016.

[48] Aditya K. Sood and Sherali Zeadally. Drive-by download attacks: A comparative study. *IT Professional*, 18(5):18–25, 2016.

[49] Alexander Sotirov. Heap feng shui in JavaScript. In *Proceedings of the Black Hat USA*, 2007.

[50] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[51] Yusuke Suzuki. Escodegen. https://www.npmjs.com/package/escodegen.

[52] PyTorch Core Team. Pytorch. https://pytorch.org/.

[53] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 269–280, 2014.

[54] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. IFuzzer: An evolutionary interpreter fuzzer using genetic programming. In *Proceedings of the European Symposium on Research in Computer Security*, pages 581–601, 2016.

[55] Dmitry Vyukov. syzkaller. https://github.com/google/syzkaller.

[56] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 579–594, 2017.

[57] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 511–522, 2013.

[58] Michal Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[59] ZERODIUM. Zerodium payouts. https://zerodium.com/program.html.