Compiler Fuzzing through Deep Learning

Chris Cummins,
Pavlos Petoumenos, Hugh Leather
University of Edinburgh
United Kingdom
{c.cummins,ppetoume,hleather}@inf.ed.ac.uk

Alastair Murray
Codeplay Software
Edinburgh, United Kingdom
alastair.murray@codeplay.com

ABSTRACT

Random program generation — fuzzing — is an effective technique for discovering bugs in compilers but successful fuzzers require extensive development effort for every language supported by the compiler, and often leave parts of the language space untested.

We introduce DeepSmith, a novel machine learning approach to accelerating compiler validation through the inference of generative models for compiler inputs. Our approach *infers* a learned model of the structure of real world code based on a large corpus of open source code. Then, it uses the model to automatically generate tens of thousands of realistic programs. Finally, we apply established differential testing methodologies on them to expose bugs in compilers. We apply our approach to the OpenCL programming language, automatically exposing bugs with little effort on our side. In 1,000 hours of automated testing of commercial and open source compilers, we discover bugs in all of them, submitting 67 bug reports. Our test cases are on average two orders of magnitude smaller than the state-of-the-art, require 3.03× less time to generate and evaluate, and expose bugs which the state-of-the-art cannot. Our random program generator, comprising only 500 lines of code, took 12 hours to train for OpenCL versus the state-of-the-art taking 9 man months to port from a generator for C and 50,000 lines of code. With 18 lines of code we extended our program generator to a second language, uncovering crashes in Solidity compilers in 12 hours of automated testing.

CCS CONCEPTS

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

KEYWORDS

Deep Learning; Differential Testing; Compiler Fuzzing.

ACM Reference Format:
Chris Cummins, Pavlos Petoumenos, Hugh Leather and Alastair Murray. 2018. Compiler Fuzzing through Deep Learning. In *ISSTA '18: ISSTA '18: International Symposium on Software Testing and Analysis, July 16–21, 2018, Amsterdam, Netherlands.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3213846.3213848

ISSTA '18, July 16–21, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5699-2/18/07…$15.00
https://doi.org/10.1145/3213846.3213848

1 INTRODUCTION

Compilers should produce correct code for valid inputs, and meaningful errors for invalid inputs. Failure to do so can hinder software development or even cause catastrophic runtime errors. Still, properly testing compilers is hard. Modern optimizing compilers are large and complex programs, and their input space is huge. Hand designed suites of test programs, while important, are inadequate for covering such a large space and will not touch all parts of the compiler.

Random test case generation — *fuzzing* — is a well established and effective method for identifying compiler bugs [6, 7, 16]. When fuzzing, randomly generated valid or semi-valid inputs are fed to the compiler. Any kind of unexpected behavior, including crashes, freezes, or wrong binaries, indicates a compiler bug. While crashes and freezes in the compiler are easy to detect, determining that binaries are correctly compiled is not generally possible without either developer provided validation for the particular program's behavior or a gold standard compiler from which to create reference outputs. In the absence of those, Differential Testing [22] can be used. The generated code and a set of inputs form a *test case* which is compiled and executed on multiple *testbeds*. If the test case should have deterministic behavior, but the output differs between testbeds, then a bug has been discovered.

Compiler fuzzing requires efficiently generating test cases that trigger compiler bugs. The state-of-the-art approach, CSmith [32], generates large random programs by defining and sampling a probabilistic grammar which covers a subset of the C programming language. Through this grammar, CSmith ensures that the generated code easily passes the compiler front-end and stresses the most complex part of the compiler, the middle-end. Complex static and dynamic analyses make sure that programs are free from undefined behavior. The programs are then differentially tested.

While CSmith has been successfully used to identify hundreds of bugs in compilers, it and similar approaches have a significant drawback. They represent a huge undertaking and require a thorough understanding of the target programming language. CSmith was developed over the course of years, and consists of over 41k lines of handwritten C++ code. By tightly coupling the generation logic with the target programming language, each feature of the grammar must be painstakingly and expertly engineered for each new target language. For example, lifting CSmith from C to OpenCL [20] — a superficially simple task — took 9 months and an additional 8k lines of code. Given the difficulty of defining a new grammar, typically only a subset of the language is implemented.

What we propose is a fast, effective, and low effort approach to the generation of random programs for compiler fuzzing. Our methodology uses recent advances in deep learning to automatically construct probabilistic models of how humans write code, instead of painstakingly defining a grammar to the same end. By training a deep neural network on a corpus of handwritten code, it is able to infer both the syntax and semantics of the programming language and the common constructs and patterns. Our approach essentially frames the generation of random programs as a language modeling problem. This greatly simplifies and accelerates the process. The expressiveness of the generated programs is limited only by what is contained in the corpus, not the developer's expertise or available time. Such a corpus can readily be assembled from open source repositories.

In this work we primarily target OpenCL, an open standard for programming heterogeneous systems, though our approach is largely language agnostic. We chose OpenCL for three reasons: it is an emerging standard with the challenging promise of functional portability across a diverse range of heterogeneous hardware; OpenCL is compiled "online", meaning that even compiler crashes and freezes may not be discovered until a product is deployed to customers; and there is already a hand written random program generator for the language to compare against. We provide preliminary results supporting DeepSmith's potential for multi-lingual compiler fuzzing.

We make the following contributions:

- a novel, automatic, and fast approach for the generation of expressive random programs for compiler fuzzing. We *infer* programming language syntax, structure, and use from real-world examples, not through an expert-defined grammar. Our system needs two orders of magnitude less code than the state-of–the-art, and takes less than a day to train;
- we discover a similar number of bugs as the state-of–the-art, but also find bugs which prior work cannot, covering more components of the compiler;
- in modeling real handwritten code, our test cases are more interpretable than other approaches. Average test case size is two orders of magnitude smaller than state-of-the-art, without any expensive reduction process.

2 DEEPSMITH

DeepSmith[1] is our open source framework for compiler fuzzing. Figure 1 provides a high-level overview. In this work we target OpenCL, though the approach is language agnostic. This section describes the three key components: a generative model for random programs, a test harness, and voting heuristics for differential testing.

2.1 Generative Model

Generating test cases for compilers is hard because their inputs are highly structured. Producing text with the right structure requires expert knowledge and a significant engineering effort, which has to be repeated from scratch for each new language. Instead, we treat the problem as an unsupervised machine learning task, employing state-of-the-art deep learning techniques to build models for how humans write programs. Our approach is inspired by breakthrough results in modeling challenging and high dimensional datasets through unsupervised learning [4, 27, 28]. Contrary to existing tools, our approach does not require expert knowledge of the target language and is only a few hundred lines of code.

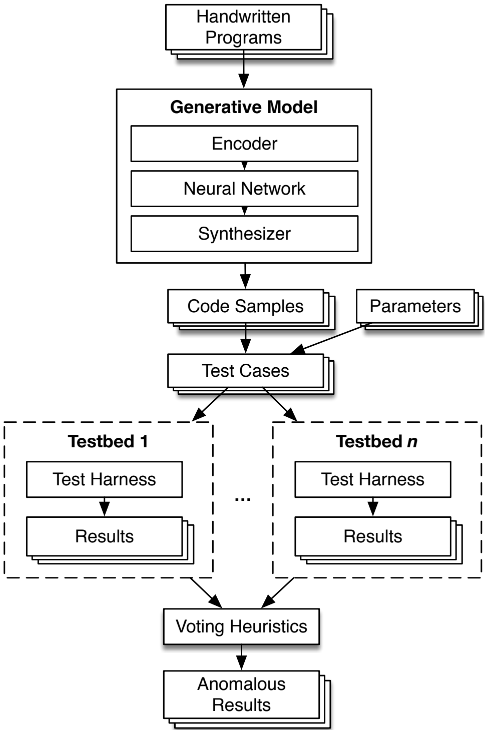

Figure 1: DeepSmith system overview.

Handwritten Programs. The generative model needs to be trained on a *seed corpus* of example programs. We automated the assembly of this corpus by mining 10k OpenCL kernels from open source repositories on GitHub. We used an *oracle compiler* (LLVM 3.9) to statically check the source files, discarding files that are not well-formed. The main purpose of this step is to remove the need to manually check that each file selected from GitHub does indeed contain OpenCL. A downside is that any training candidate which triggers a bug in the LLVM 3.9's front end will not be included. However, this did not prevent our system from uncovering errors in that compiler (Section 4.4).

This corpus, exceeding one million lines of code, is used as a representative sample of OpenCL code from which a generative model can be derived.

Encoder. The textual representation of program codes must be encoded as numeric sequences for feeding as input to the machine learning model. Prior machine learning works have used character-level encodings, token-level encodings, or fixed length feature vectors. We extend the hybrid character/token-level encoding of [9], in which a programming language's keywords and common names are treated as individual tokens while the rest of the text is encoded on a character-level basis. This approach hits a balance between compressing the input text and keeping the number of tokens in the vocabulary low.

[1] DeepSmith available at: https://chriscummins.cc/deepsmith

We additionally employed semantic-preserving transformations to simplify the training programs. First, each source file is preprocessed to expand macros and remove conditional compilation and comments. Then, all user-declared identifiers are renamed using an arbitrary, but consistent pattern based on their order of declaration: $\{a, b, c, \ldots, aa, ab, ac, \ldots\}$ for variables and $\{A, B, C, \ldots, AA, AB, AC, \ldots\}$ for functions. This ensures a consistent naming convention, without modifying program behavior. Finally, a uniform code style is enforced to ensure consistent use of braces, parentheses, and white space. These rewriting simplifications give more opportunities for the model to learn the structure and deeper aspects of the language and speed up the learning. On the other hand, some bugs in the preprocessor or front-end might no longer be discoverable. We reason that this is an acceptable trade-off. For languages where the corpus can be many orders of magnitude larger, for example, C or Java, models may be generated without these modifications.

Neural Network. We use the Long Short-Term Memory (LSTM) architecture of Recurrent Neural Network to model program code [12]. In the LSTM architecture activations are learned with respect not just to their current inputs but to previous inputs in a sequence. In our case, this allows modeling the probability of a token appearing in the text given a history of previously seen tokens. Unlike previous recurrent networks, LSTMs employ a *forget gate* with a linear activation function, allowing them to avoid the *vanishing gradients* problem [24]. This makes them effective at learning complex relationships over long sequences [21] which is important for modeling program code. Our LSTM networks model the vocabulary distribution over the encoded corpus. After initial experiments using different model parameters, we found that a two layer LSTM network of 512 nodes per layer provided a good trade-off between the fidelity of the learned distribution and the size of the network, which limits the rate of training and inference. The network is trained using Stochastic Gradient Descent for 50 epochs, with an initial learning rate of 0.002 and decaying by 5% every epoch. Training the model on the OpenCL corpus took 12 hours using a single NVIDIA Tesla P40. We provided the model with no prior knowledge of the structure or syntax of a programming language.

Program Generation. The trained network is sampled to generate new programs. The model is seeded with the start of a kernel (identified in OpenCL using the keywords `kernel void`), and sampled token-by-token. A "bracket depth" counter is incremented or decremented upon production of `{` or `}` tokens respectively, so that the end of the kernel can be detected and sampling halted. The generated sequence of tokens is then decoded back to text and used for compiler testing.

2.2 Test Harness

OpenCL is an embedded compute kernel language, requiring host code to compile, execute, and transfer data between the host and device. For the purpose of compiler fuzzing, this requires a *test harness* to run the generated OpenCL programs. At first, we used the test harness of CLSmith. The harness assumes a kernel with no input and a `ulong` buffer as its single argument where the result is written. Only 0.2% of the GitHub kernels share this structure. We desired a more flexible harness so as to test a more expressive range of programs, capable of supporting multi-argument kernels and generating data to use as inputs.

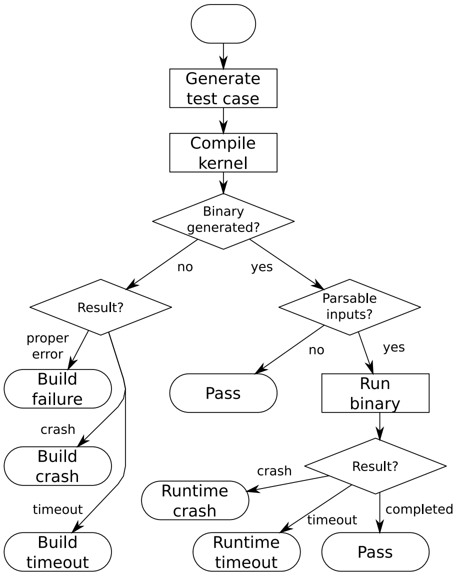

Figure 2: Test case execution, and possible results.

We developed a harness which first determines the expected arguments from the function prototype and generates host data for them. At the moment, we support scalars and arrays of all OpenCL primitive and vector types. For a kernel execution across n threads, buffers of size n are allocated for pointer arguments and populated with values $[1 \ldots n]$; scalar inputs are given value n, since we observe that most kernels use these for specifying buffer sizes.

The training programs from which the generative model is created are real programs, and as such do not share the argument type restrictions. The model, therefore, may generate correct programs for which our driver cannot create example inputs. In this case, a "compile-only" stub is used, which only compiles the kernel, without generating input data or executing the compiled kernel.

Unlike the generative model, this test harness is language-specific and the design stems from domain knowledge. Still, it is a relatively simple procedure, consisting of a few hundred lines of Python.

Test Harness Output Classes. Executing a test case on a testbed leads to one of seven possible outcomes, illustrated in Figure 2. A *build failure* occurs when online compilation of the OpenCL kernel fails, usually accompanied by an error diagnostic. A *build crash* or *build timeout* outcome occurs if the compiler crashes or fails to produce a binary within 60 seconds, respectively. For compile-only test cases, a *pass* is achieved if the compiler produces a binary. For test cases in which the kernel is executed, kernel execution leads to one of three potential outcomes: *runtime crash* if the program crashes, *timeout* if the kernel fails to terminate within 60 seconds, or *pass* if the kernel terminates gracefully and computes an output.

#.	Platform	Device	Driver	OpenCL	Operating system	Device Type	Open Source?	Bug Reports Submitted
1	NVIDIA CUDA	GeForce GTX 1080	375.39	1.2	Ubuntu 16.04 64bit	GPU		8
2	NVIDIA CUDA	GeForce GTX 780	361.42	1.2	openSUSE 13.1 64bit	GPU		1
3	Beignet	Intel HD Haswell GT2	1.3	1.2	Ubuntu 16.04 64bit	GPU	Yes	13
4	Intel OpenCL	Intel E5-2620 v4	1.2.0.25	2.0	Ubuntu 16.04 64bit	CPU		6
5	Intel OpenCL	Intel E5-2650 v2	1.2.0.44	1.2	CentOS 7.1 64bit	CPU		1
6	Intel OpenCL	Intel i5-4570	1.2.0.25	1.2	Ubuntu 16.04 64bit	CPU		5
7	Intel OpenCL	Intel Xeon Phi	1.2	1.2	CentOS 7.1 64bit	Accelerator		3
8	POCL	POCL (Intel E5-2620)	0.14	1.2	Ubuntu 16.04 64bit	CPU	Yes	22
9	Codeplay	ComputeAorta (Intel E5-2620)	1.14	1.2	Ubuntu 16.04 64bit	CPU		1
10	Oclgrind	Oclgrind Simulator	16.10	1.2	Ubuntu 16.04 64bit	Emulator	Yes	7

Table 1: OpenCL systems and the number of bug reports submitted to date (22% of which have been fixed, the remainder are pending). For each system, two testbeds are created, one with compiler optimizations, the other without.

2.3 Voting Heuristics for Differential Testing

We employ established Differential Testing methodologies to expose compiler defects. As in prior work, voting on the output of programs across compilers has been used to circumvent the *oracle problem* and detect miscompilations [22]. However, we extend this approach to describe not only miscompilations, but also anomalous build failures and crashes.

When evaluating the outcomes of test cases, build crash (**bc**) and build timeout (**bto**) outcomes are of immediate interest, indicative of erroneous compiler behavior (examples may be found in Section 4.1). For all other outcomes, *differential tests* are required to confirm anomalous behavior. We look for test cases where there is a majority outcome – i.e. for which some fraction of the testbeds behave the same – but some testbed deviates. We use the presence of the majority increasing the likelihood that there is a 'correct' behavior for the test case. In this work, we choose the majority fraction to be $\lceil \frac{2}{3}n \rceil$, where n is the number of testbeds.

An *anomalous build failure* (**abf**) or *anomalous runtime crash* (**arc**) occurs if, for a given test case, the majority of testbeds execute successfully, and a testbed yields a compilation error or runtime crash. An *anomalous wrong-output* (**awo**) occurs if, for a given test case, the majority of testbeds execute successfully, producing the same output values, and a testbed yields a result which differs from this majority output. Anomalous wrong-output results are indicative of *miscompilations*, a particularly hard to detect class of bug in which the compiler silently emits wrong code. CSmith is designed specifically to target this class of bug.

False Positives for Anomalous Runtime Behavior. Generated programs may contain undefined or non-deterministic behavior which will incorrectly be labeled as anomalous. CSmith circumvents this problem by performing complex analyses during generation so as to minimize the chance of producing programs with undefined behavior. Although similar analyses could be created as filters for our system, we take a simpler approach, filtering only the few types of non-deterministic behavior we have actually observed to happen in practice.

We filter data races, out-of-bounds and uninitialized accesses with GPUverify [2] and Oclgrind [26]. Some compiler warnings provide strong indication of non-deterministic behavior (e.g. comparison between pointer and integer) – we check for these warnings and filter accordingly.

Floating point operations in OpenCL can be imprecise, so code can produce different output on different testbeds. For this reason, CSmith and CLSmith do not support floating point operations. DeepSmith allows floating point operations but since it cannot apply differential testing on the outputs, it can detect all results except for the *anomalous wrong-output* results.

The last type of undefined behavior we observed comes from division by zero and related mathematical functions which require non-zero values. We apply a simple detection and filtering heuristic – we change the input values and check to see if the output remains anomalous. While theoretically insufficient, in practice we found that no false positives remained.

3 EXPERIMENTAL SETUP

In this section we describe the experimental parameters used.

3.1 OpenCL Systems

We conducted testing of 10 OpenCL systems, summarized in Table 1. We covered a broad range of hardware: 3 GPUs, 4 CPUs, a co-processor, and an emulator. 7 of the compilers tested are commercial products, 3 of them are open source. Our suite of systems includes both combinations of different drivers for the same device, and different devices using the same driver.

3.2 Testbeds

For each OpenCL system, we create two testbeds. In the first, the compiler is run with optimizations disabled. In the second, optimizations are enabled. Each testbed is then a triple, consisting of *<device, driver, is_optimized>* settings. This mechanism gives 20 testbeds to evaluate.

3.3 Test Cases

For each generated program we create inputs as described in Section 2.2. In addition, we need to choose the number of threads to use. We generate two test cases, one using one thread, the other using 2048 threads. A test case is then a triple, consisting of *<program, inputs, threads>* settings.

3.4 Bug Search Time Allowance

We compare both our fuzzer and CLSmith. We allow both to run for 48 hours on each of the 20 testbeds. CLSmith used its default configuration. The total runtime for a test case consists of the generation and execution time.

```
kernel void A(global float* a, global float* b) {
  a[0] = max(a[c], b[2]);
}
```

(a) Testbeds 10± assertion *Uncorrected typos!* during semantic analysis.

```
kernel void A(float4 a, global float4* b,
              global float4* c, unsigned int d,
              global double* e, global int2* f,
              global int4* g, constant int* h,
              constant int* i) {
  A(a, b, c, d, d, e, f, g, h);
}
```

(b) Testbeds 1±, 2± segmentation fault due to implicit address space conversion.

```
kernel void A(read_only image2d_t a,
              global double2* b) {
  b[0] = get_global_id(0);
}
```

(c) Testbeds 3± assertion *sel.hasDoubleType()* during code generation.

```
kernel void A(global float4* a) {
  a[get_local_id(0) / 8][get_local_id(0)] =
      get_local_id(0);
}
```

(d) Testbeds 3± assertion *scalarizeInsert* during code generation.

```
kernel void A() {
  __builtin_astype(d, uint4);
}
```

(e) Of the 10 compilers we tested, 6 crash with segfault when compiling this kernel.

Figure 3: Example kernels which crash compilers.

4 EVALUATION

We report on the results of DeepSmith testing of the 10 OpenCL systems from Table 1, in which each ran for 48 hours. We found bugs in all the compilers we tested — every compiler crashed, and every compiler generated programs which either crash or silently compute the wrong result. To date, we have submitted 67 bug reports to compiler vendors. We first provide a qualitative analysis of compile-time and runtime defects found, followed by a quantitative comparison of our approach against the state-of-the-art in OpenCL compiler fuzzing — CLSmith [20]. DeepSmith is able to identify a broad range of defects, many of which CLSmith cannot, for only a fraction of the engineering effort. Finally, we provide a quantitative analysis of compiler robustness over time, using the compiler crash rate of every LLVM release in the past two years as a metric of compiler robustness. We find that progress is good, compilers are becoming more robust, yet the introduction of new features and regressions ensures that compiler validation remains a moving target.

Unless stated otherwise, DeepSmith code listings are presented verbatim, with only minor formatting changes applied to save space. No test case reduction, either manual or automatic, was needed.

For the remainder of the paper we identify testbeds using the OpenCL system number from Table 1, suffixed with +, −, or ± to denote optimizations on, off, or either, respectively.

```
void A(){(global a*)()}
```

(a) Reduced from 48 line kernel.

```
void A(){void* a; uint4 b=0; b=(b>b)?a:a}
```

(b) Reduced from 52 line kernel.

```
void A(){double2 k; return (float4)(k,k,k,k)}
```

(c) Reduced from 68 line kernel.

Figure 4: Example codes which crash parsers.

4.1 Compile-time Defects

OpenCL is typically compiled online, which amplifies the significance of detecting compile-time defects, as they may not be discovered until code has been shipped to customers. We found numerous cases where DeepSmith kernels trigger a crash in the compiler (and as a result, the host process), or cause the compiler to loop indefinitely. In the testing time allotted we have identified 199 test cases which trigger unreachable code failures, triggered 31 different compiler assertions, and produced 114 distinct stack traces from other compiler crashes.

Semantic Analysis Failures. Compilers should produce meaningful diagnostics when inputs are invalid, yet we discovered dozens of compiler defects attributable to improper or missing error handling. Many generation and mutation based approaches to compiler validation have focused solely on testing under *valid inputs*. As such, this class of bugs may go undiscovered. We believe that our approach contributes a significant improvement to generating plausibly-erroneous code over prior random-enumeration approaches.

The use of undeclared identifiers is a core error diagnostic which one would expect to be robust in a mature compiler. DeepSmith discovered cases in which the presence of undeclared identifiers causes the Testbeds 10± compiler to crash. For example, the undeclared identifier c in Figure 3a raises an assertion during semantic analysis of the AST when used as an array index.

Type errors were an occasional cause of compile-time defect. Figure 3b induces a crash in NVIDIA compilers due to an implicit conversion between global to constant address qualifiers. Worse, we found that Testbeds 3± would loop indefinitely on some kernels containing implicit conversions from a pointer to an integer, as shown in Figure 5a. While spinning, the compiler would utilize 100% of the CPU and consume an increasing amount of host memory until the entire system memory is depleted and the process crashes.

Occasionally, incorrect program semantics will remain undetected until late in the compilation process. Both Figures 3c and 3d pass the type checker and semantic analysis, but trigger compiler assertions during code generation.

An interesting yet unintended byproduct of having trained DeepSmith on thousands of real world examples is that the model learned to occasionally generate compiler-specific code, such as invoking compiler builtins. We found the quality of error handling on these builtins to vary wildly. For example, Figure 3e silently crashes 6 of the 10 compilers, which, to the best of our knowledge, makes

```
kernel void A(global int* a) {
  int b = get_global_id(0);
  a[b] = (6 * 32) + 4 * (32 / 32) + a;
}
```

(a) Testbeds 3± loop indefinitely, leaking memory until the entire system memory is depleted and the process crashes.

```
kernel void A(global float* a, global float* b,
              global float* c) {
  int d, e, f;
  d = get_local_id(0);
  for (int g = 0; g < 100000; g++)
    barrier(1);
}
```

(b) Testbed 1+ hangs during optimization of kernels with large loop bounds. Testbeds 1− and 2± compile in under 1 second.

```
kernel void A(global int* a) {
  int b = get_global_id(0);
  while (b < 512) { }
}
```

(c) Testbeds 4+, 5+, 6+, 7+ hang during optimization of kernels with non-terminating loops.

```
kernel void A(global unsigned char* a,
              unsigned b) {
  a[get_global_id(0)] %= 42;
  barrier(1);
}
```

(d) Testbeds 7± loops indefinitely, consuming 100% CPU usage.

Figure 5: Example kernels which hang compilers.

DeepSmith the first random program generator to induce a defect through exploiting compiler-specific functionality.

Parser Failures. Parser development is a mature and well understood practice. We uncovered parser errors in several compilers. Each of the code samples in Figure 4 induce crash errors during parsing of compound statements in both Testbeds 5± and 7±. For space, we have hand-reduced the listings to minimal code samples, which we have reported to Intel. Each reduction took around 6 edit-compile steps, taking less than 10 minutes. In total, we have generated 100 distinct programs which crash compilers during parsing.

Compiler Hangs. As expected, some compile-time defects are optimization sensitive. Testbed 1+ hangs on large loop bounds, shown in Figure 5b. All commercial Intel compilers we tested hang during optimization of non-terminating loops (Figure 5c).

Testbeds 7± loop indefinitely during compilation of the simple kernel in Figure 5d.

Other errors. Some compilers are more permissive than others. Testbeds 4±, 6±, 9± reject out-of-range literal values e.g. `int i = 0xFFFFFFFFFFFFFFFFFFFFFFFFF`, whilst Testbeds 3±, 5±, 7±, 8±, and 10± interpret the literal as an `unsigned long long` and implicitly cast to an integer value of -1. Testbeds 1±, 2± emit no warning.

Testbeds 1±, 2±, 3± rejected address space qualifiers on automatic variables, where all other testbeds successfully compiled and executed.

```
kernel void A(global double* a, global double* b,
              global double* c, int d, int e) {
  double f;
  int g = get_global_id(0);
  if (g < e - d - 1)
    c[g] = (((e) / d) % 5) % (e + d);
}
```

(a) Testbeds 4+, 6+ incorrectly optimize the `if` statement, causing the conditional branch to execute (it shouldn’t). This pattern of integer comparison to thread ID is widely used.

```
kernel void A(global int* a, global int* b) {
  switch (get_global_id(0)) {
  case 0:
    a[get_global_id(0)]=b[get_global_id(0)+13];
    break;
  case 2:
    a[get_global_id(0)]=b[get_global_id(0)+11];
    break;
  case 6:
    a[get_global_id(0)]=b[get_global_id(0)+128];
  }
  barrier(2);
}
```

(b) A race condition in `switch` statement evaluation causes 10± to sporadically crash when executed with a number of threads > 1.

```
kernel void A(global int* a, global int* b,
              global int* c) {
  c[0] = (a[0] > b[0]) ? a[0] : 0;
  c[2] = (a[3] <= b[3]) ? a[4] : b[5];
  c[4] = (a[4] <= b[5]) ? a[7] : b[7];
  c[7] = (a[7] < b[0]) ? a[0] : (a[0] > b[1]);
}
```

(c) Testbeds 3± silently miscompile ternary assignments in which the operands are different global buffers.

```
kernel void A(local int* a) {
  for (int b = 0; b < 100; b++)
    B(a);
}
```

(d) Compilation should fail due to call to undefined function `B()`; Testbeds 8± silently succeed then crash upon kernel execution.

Figure 6: Example kernels which are miscompiled.

On Testbeds 3±, the statement `int n = mad24(a, (32), get_global_size(0));` (a call to a math builtin with mixed types) is rejected as ambiguous.

4.2 Runtime Defects

Prior work on compiler test case generation has focused on extensive stress-testing of compiler middle-ends to uncover miscompilations [6]. CSmith, and by extension, CLSmith, specifically targets this class of bugs. Grammar based enumeration is highly effective at this task, yet is bounded by the expressiveness of the grammar. Here we provide examples of bugs which cannot currently be discovered by CLSmith.

Thread-dependent Flow Control. A common pattern in OpenCL is to obtain the thread identity, often as an `int`, and to compare this against some fixed value to determine whether or not to complete a unit of work (46% of OpenCL kernels on GitHub use this (tid → int, `if (tid < ...) {...}`) pattern). DeepSmith, having modeled the frequency with which this pattern occurs in real handwritten code, generates many permutations of this pattern. And in doing so, exposed a bug in the optimizer of Testbeds 4+ and 6+ which causes the `if` branch in Figure 6a to be erroneously executed when the kernel is compiled with optimizations enabled. We have reported this issue to Intel. CLSmith does not permit the thread identity to modify control flow, rendering such productions impossible.

Figure 6b shows a simple program in which thread identity determines the program output. We found that this test case would sporadically crash Testbeds 10±, an OpenCL device simulator and debugger. Upon reporting to the developers, the underlying cause was quickly diagnosed as a race condition in `switch` statement evaluation, and fixed within a week.

Kernel Inputs. CLSmith kernels accept a single buffer parameter into which each thread computes its result. This fixed prototype limits the ability to detect bugs which depend on input arguments. Figure 6c exposes a bug of this type. Testbeds 3± will silently miscompile ternary operators when the ternary operands consist of values stored in multiple different global buffers. CLSmith, with its fixed single input prototype, is unable to discover this bug.

Latent Compile-time Defects. Sometimes, invalid compiler inputs may go undetected, leading to runtime defects only upon program execution. Since CLSmith enumerates only well-formed programs, this class of bugs cannot be discovered.

Figure 6d exposes a bug in which a kernel containing an undefined symbol will successfully compile without warning on Testbeds 8±, then crash the program when attempting to run the kernel. This issue has been reported to the developers and fixed.

4.3 Comparison to State-of-the-art

In this section, we provide a quantitative comparison of the bug-finding capabilities of DeepSmith and CLSmith.

Results Overview. Table 2 shows the results of 48 hours of consecutive testing for all Testbeds. An average of 15k CLSmith and 91k DeepSmith test cases were evaluated on each Testbed, taking 12.1s and 1.90s per test case respectively. There are three significant factors providing the sixfold increase in testing throughput achieved by DeepSmith over CLSmith: test cases are faster to generate, test cases are less likely to timeout (execute for 60 seconds without termination), and the test cases which do not timeout execute faster.

Figure 7a shows the generation and execution times of DeepSmith and CLSmith test cases, excluding timeouts[2]. DeepSmith generation time grows linearly with program length, and is on average 2.45× faster than CLSmith. Test case execution is on average 4.46× faster than CLSmith.

The optimization level generally does not affect testing throughput significantly, with the exception of Testbed 7+. Optimization of large structs is expensive on Testbed 7+, and CLSmith test cases use global structs extensively. This is a known issue — in [20] the authors omit large-scale testing on this device for this reason. The use of structs in handwritten OpenCL is comparatively rare — only 7.1% of kernels on GitHub use them.

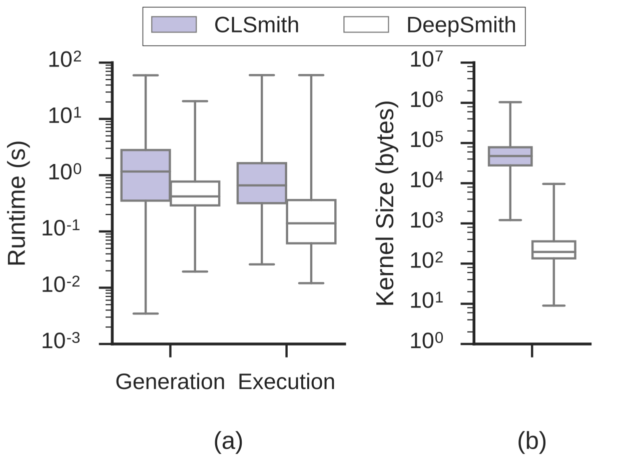

Figure 7: Comparison of runtimes (a) and test case sizes (b). DeepSmith test cases are on average evaluated 3.03× faster than CLSmith (2.45×, and 4.46× for generation and execution, respectively), and are two orders of magnitude smaller. Timings do not include the cost of timeouts which would increase the performance gains of DeepSmith by nearly a factor of two.

Comparison of Test Cases. The average CLSmith program is 1189 lines long (excluding headers). CLSmith test cases require reduction in order to expose the underlying bug. An automated approach to OpenCL test case reduction is presented in [25], though it requires on average 100 minutes for each test case using a parallelized implementation (and over 6 hours if this parallelization is not available); the authors also suggest a final manual pass after automated reduction. In contrast, DeepSmith learned to program from humans, and humans do not typically write such large kernel functions. The average DeepSmith kernel is 20 lines long, which is interpretable without reduction, either manual or automatic.

Comparison of Results. Both testing systems found anomalous results of all types. In 48 hours of testing, CLSmith discovered compile-time crashes (**bc**) in 8 of the 20 testbeds, DeepSmith crashed all of them. DeepSmith triggered 31 distinct compiler assertions, CLSmith 2. Both of the assertions triggered by CLSmith were also triggered by DeepSmith. DeepSmith also triggered 3 distinct *unreachable!* compile-time crashes, CLSmith triggered 0. The ratio of build failures is higher in the token-level generation of DeepSmith (51%) than the grammar-based generation of CLSmith (26%).

The Intel CPU Testbeds (4±, 5±, 6±, and 7±) would occasionally emit a stack trace upon crashing, identifying the failure point in a specific compiler pass. CLSmith triggered such crashes in 4 distinct passes. DeepSmith triggered crashes in 10 distinct passes, including 3 of the 4 in which CLSmith did. Figure 8 provides examples. Many of these crashes are optimization sensitive, and are more likely to occur when optimizations are enabled. CLSmith was able to induce a crash in only one of the Intel testbeds with optimizations disabled. DeepSmith crashed all of the compilers with both optimizations enabled and disabled.

[2] If timeouts are included then the performance improvement of DeepSmith is 6.5× with the execution times being 11× faster. However, this number grows as we change the arbitrary timeout threshold, so for fairness we have chosen to exclude it.

			CLSmith							DeepSmith						
#.	**Device**	**±**	**bc**	**bto**	**abf**	**arc**	**awo**	**✓**	**total**	**bc**	**bto**	**abf**	**arc**	**awo**	**✓**	**total**
1	GeForce GTX 1080	−	0	0	0	2	2	15628	15632	27	0	3	0	5	62105	62140
		+	0	71	0	6	9	14007	14093	20	1	1	0	7	57361	57390
2	GeForce GTX 780	−	0	0	0	28	5	18220	18253	27	0	3	0	9	87129	87168
		+	26	14	0	0	3	17654	17697	32	1	1	0	9	82666	82709
3	Intel HD Haswell GT2	−	2714	2480	0	0	3	1121	6318	574	200	2	0	12	136977	137765
		+	2646	2475	0	0	3	1075	6199	569	200	5	0	10	135430	136214
4	Intel E5-2620 v4	−	0	27	1183	0	0	16313	17523	57	0	9	1	0	107982	108049
		+	487	87	1130	0	0	17350	19054	320	147	7	3	0	113616	114093
5	Intel E5-2650 v2	−	0	11	0	0	0	17887	17898	152	2	0	0	0	90882	91036
		+	112	175	0	0	0	14626	14913	170	117	0	0	1	90478	90766
6	Intel i5-4570	−	0	14	1226	0	0	17118	18358	73	0	9	2	1	111240	111325
		+	526	63	1180	0	0	19185	20954	318	140	7	2	1	117049	117517
7	Intel Xeon Phi	−	4	84	0	0	8	13265	13361	68	4	0	0	1	37171	37244
		+	42	1474	0	0	2	3258	4776	77	47	0	0	0	37501	37625
8	POCL (Intel E5-2620)	−	0	0	0	675	0	17250	17925	54	1	2	89	3	85318	85467
		+	0	3	0	99	5	13980	14087	46	0	1	104	4	81267	81422
9	ComputeAorta (Intel E5-2620)	−	0	0	0	0	0	18479	18479	51	0	1	3	1	112324	112380
		+	0	0	0	300	11	18625	18936	59	0	0	48	4	115323	115434
10	Oclgrind Simulator	−	0	0	0	0	0	5287	5287	2081	0	0	0	1	73261	75343
		+	0	0	0	0	0	5334	5334	2265	0	0	0	0	77959	80224

Table 2: Results from 48 hours of testing using CLSmith and DeepSmith. System #. as per Table 1. ± denotes optimizations off (−) vs on (+). The remaining columns denote the number of build crash (bc), build timeout (bto), anomalous build failure (abf), anomalous runtime crash (arc), anomalous wrong-output (awo), and pass (✓) results.

CLSmith produced many **bto** results across 13 Testbeds. Given the large kernel size, it is unclear how many of those are infinite loops or simply a result of slow compilation of large kernels. The average size of CLSmith **bto** kernels is 1558 lines. Automated test case reduction — in which thousands of permutations of a program are executed — may be prohibitively expensive for test cases with very long runtimes. DeepSmith produced **bto** results across 11 Testbeds and with an average kernel size of 9 lines, allowing for rapid identification of the underlying problem.

The integrated GPU Testbeds (3±) frequently failed to compile CLSmith kernels, resulting in over 10k **bc** and **bto** results. Of the build crashes, 68% failed silently, and the remainder were caused by the same two compiler assertions for which DeepSmith generated 4 line test cases, shown in Figure 9. DeepSmith also triggered silent build crashes in Testbeds 3±, and a further 8 distinct compiler assertions.

The 4719 **abf** results for CLSmith on Testbeds 4± and 6± are all a result of compilers rejecting empty declarations, (e.g. `int;`) which CLSmith occasionally emits. DeepSmith also generated these statements, but with a much lower probability, given that it is an unusual construct (0.6% of test cases, versus 7.0% of CLSmith test cases).

ComputeAorta (Testbeds 9±) defers kernel compilation so that it can perform optimizations dependent on runtime parameters. This may contribute to the relatively large number of **arc** results and few **bc** results of Testbeds 9±. Only DeepSmith was able to expose compile-time defects in this compiler.

Over the course of testing, a combined 3.4×10^8 lines of CLSmith code was evaluated, compared to 3.8×10^6 lines of DeepSmith code. This provides CLSmith a greater potential to trigger miscompilations. CLSmith generated 33 programs with anomalous wrong-outputs. DeepSmith generated 30.

4.4 Compiler Stability Over Time

The Clang front-end to LLVM supports OpenCL, and is commonly used in OpenCL drivers. This in turn causes Clang-related defects to potentially affect multiple compilers, for example the one in Figure 3e. To evaluate the impact of Clang, we used debug+assert builds of every LLVM release in the past 24 months and processed 75,000 DeepSmith kernels through the Clang front-end (this includes the lexer, parser, and type checker, but not code generation).

Figure 10 shows that the crash rate of the Clang front-end is, for the most part, steadily decreasing over time. The number of failing compiler crashes decreased tenfold between 3.6.2 and 5.0.0. Table 3 shows the 7 distinct assertions triggered during this experiment. Assertion 1 (*Uncorrected typos!*) is raised on all compiler versions — see Figure 3a for an example. The overall rate at which the assertion is triggered has decreased markedly, although there are slight increases between some releases. Notably, the current development trunk has the second lowest crash rate, but is joint first in terms of the number of unique assertions. Assertions 3 (*Addr == 0 || hasTargetSpecificAddressSpace()*) and 4 (*isScalarType()*) were triggered by some kernels in the development trunk but not under any prior release. We have submitted bug reports for each of the three assertions triggered in the development trunk, as well as for two distinct unreachables.

The results emphasize that compiler validation is a moving target. Every change and feature addition has the potential to introduce regressions or new failure cases. Since LLVM will not release unless their compiler passes their own extensive test suites, this also reinforces the case for compiler fuzzing. We believe our approach provides an effective means for the generation of such fuzzers, at a fraction of the cost of existing techniques.

```
kernel void A() {
  while (true)
    barrier(1);
}
```

(a) ***Post-Dominance Frontier Construction*** **pass.**

```
kernel void A(global float* a, global float* b,
              const int c) {
  for (int d = 0; d < c; d++)
    for (d = 0; d < a; d += 32)
      b[d] = 0;
}
```

(b) ***Simplify the CFG*** **pass.**

```
kernel void A(global int* a) {
  int b = get_global_id(0);
  while (b < *a)
    if (a[0] < 0)
      a[1] = b / b * get_local_id(0);
}
```

(c) ***Predicator*** **pass.**

```
kernel void A(global float* a, global float* b,
              global float* c, const int d) {
  for (unsigned int e = get_global_id(0);
       e < d; e += get_global_size(0))
    for (unsigned f = 0; f < d; ++f)
      e += a[f];
}
```

(d) ***Combine redundant instructions*** **pass.**

```
kernel void A(int a, global int* b) {
  int c = get_global_id(0);
  int d = work_group_scan_inclusive_max(c);
  b[c] = c;
}
```

(e) ***PrepareKernelArgs*** **pass.**

```
kernel void A() {
  local float a; A(a);
}
```

(f) ***Add SPIR related module scope metadata*** **pass.**

```
kernel void A() {
  local int a[10];
  local int b[16][16];
  a[1024 + (2 * get_local_id(1) +
    get_local_id(0)) + get_local_id(0)] = 6;
  barrier(b);
}
```

(g) ***Intel OpenCL RemoveDuplicationBarrier*** **pass.**

```
kernel void A(global half* a) {
  int b = get_global_id(0);
  a[b] = b * b;
}
```

(h) ***X86 DAG->DAG Instruction Selection*** **pass.**

Figure 8: Example kernels which crash Intel compiler passes.

```
kernel void A(global int* a, global int* b,
              global int* c) {
  a[get_global_id(0)] = a[get_global_id(0)] > b;
}
```

(a) Assertion ***storing/loading pointers only support private array.***

```
kernel void A(global int* a) {
  global int* b = ((void*)0);
  b[0] = a;
}
```

(b) Assertion ***iter != pointerOrigMap.end().***

Figure 9: Example kernels which trigger compiler assertions which both CLSmith and DeepSmith exposed.

4.5 Extensibility of Language Model

A large portion of the DeepSmith architecture is language-agnostic, requiring only a corpus, encoder, and harness for each new language. This potentially significantly lowers the barrier-to-entry compared with prior grammar-based fuzzers. To explore this, we report on initial results in extending DeepSmith to the Solidity programming language. Solidity is the smart contract programming language of the Ethereum blockchain. At less than four years old, it lacks much of the tooling of more established programming languages. Yet, it is an important candidate for rigorous testing, as exploitable bugs may undermine the integrity of the blockchain and lead to fraudulent transactions.

Testing Methodology. We applied the same methodology to train the program generator as for OpenCL. We assembled a corpus of Solidity contracts from GitHub, recursively inlining imported modules where possible. We used the same tokenizer as for OpenCL, only changing the list of language keywords and builtins. Code style was enforced using clang-format. We trained the model in the same manner as OpenCL. No modification to either the language model or generator code was required. We created a simple compile-only test harness to drive the generated Solidity contracts.

Initial Results. We ran the generator and harness loop for 12 hours on four testbeds: the Solidity reference compiler `solc` with optimizations on or off, and `solc-js`, which is an Emscripten compiled version of the `solc` compiler. Our results are summarized in Table 4. We found numerous cases where the compiler silently crashes, and two distinct compiler assertions. The first is caused by missing error handling of language features (this issue is known to the developers). The source of the second assertion is the JavaScript runtime and is triggered only in the Emscripten version, suggesting an error in the automatic translation from LLVM to JavaScript.

Extending DeepSmith to a second programming required an additional 150 lines of code (18 lines for the generator and encoder, the remainder for the test harness) and took about a day. Given the re-usability of the core DeepSmith components, there is a diminishing cost with the addition of each new language. For example, the OpenCL encoder and re-writer, implemented using LLVM, could be adapted to C with minimal changes. Given the low cost of extensibility, we believe these preliminary results indicate the utility of our approach for simplifying test case generation.

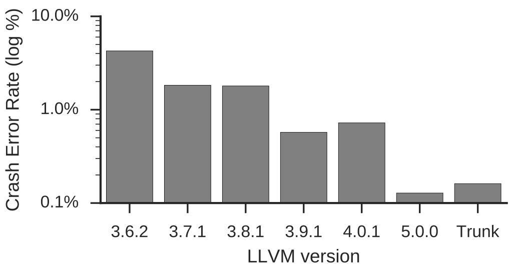

Figure 10: Crash rate of the Clang front-end of every LLVM release in the past 24 months compiling 75k DeepSmith kernels.

	3.6.2	3.7.1	3.8.1	3.9.1	4.0.1	5.0.0	Trunk
Assertion 1	2962	1327	1332	414	523	83	97
Assertion 2		1	1				
Assertion 3							1
Assertion 4							2
Assertion 5	147						
Assertion 6	1						
Assertion 7				1	1		
Unreachable	86	42	14	14	18	13	21

Table 3: The number of DeepSmith programs which trigger distinct Clang front-end assertions, and the number of programs which trigger unreachables.

5 RELATED WORK

The random generation of test cases is a well established approach to the compiler validation problem. Prior approaches are surveyed in [3, 16] and empirically contrasted in [6]. The main question of interest is in how to efficiently generate codes which trigger bugs. There are two main approaches: *program generation*, where inputs are synthesized from scratch; and *program mutation*, where existing codes are modified so as to identify anomalous behavior.

Program Generation. In the foundational work on differential testing for compilers, McKeeman *et al.* present generators capable of enumerating programs of a range of qualities, from random ASCII sequences to C model conforming programs [22]. Subsequent works have presented increasingly complex generators which improve in some metric of interest, generally expressiveness or probability of correctness. CSmith [32] is a widely known and effective generator which enumerates programs by pairing infrequently combined language features. In doing so, it produces correct programs with clearly defined behavior but very unlikely functionality, increasing the chances of triggering a bug. Achieving this required extensive engineering work, most of it not portable across languages, and ignoring some language features. Subsequent generators influenced by CSmith, like Orange3 [23], focus on features and bug types beyond the scope of CSmith, arithmetic bugs in the case of Orange3. Glade [1] derives a grammar from a corpus of example programs. The derived grammar is enumerated to produce new programs, though unlike our approach, no distribution is learned over the grammar; program enumeration is uniformly random.

Program Mutation. Equivalence Modulo Inputs (EMI) testing [19, 29] follows a different approach to test case generation. Starting with existing code, it inserts or deletes statements that will not be executed, so functionality should remain the same. If it is affected, it is due to a compiler bug. While a powerful technique able to find hard to detect bugs, it relies on having a very large number of programs to mutate. As such, it still requires an external code generator. Similarly to CSmith, EMI favors very long test programs. LangFuzz [13] also uses mutation but does this by inserting code segments which have previously exposed bugs. This increases the chances of discovering vulnerabilities in scripting language engines. Skeletal program enumeration [34] again works by transforming existing code. It identifies algorithmic patterns in short pieces of code and enumerates all the possible permutations of variable usage. Compared to all these, our fuzzing approach is low cost, easy to develop, portable, capable of detecting a wide range of errors, and focusing by design on bugs that are more likely to be encountered in a production scenario.

Compiler	±	Silent Crashes	Assertion 1	Assertion 2
solc	−	204	1	
	+	204	1	
solc-js	−	3628	1	1
	+	908	1	1

Table 4: The number of DeepSmith programs that trigger Solidity compiler crashes from 12 hours of testing.

Machine Learning. There is an increasing interest in applying machine learning to software testing. Most similar to our work is Learn&fuzz [10], in which an LSTM network is trained over a corpus of PDF files to generate test inputs for the Microsoft Edge renderer, yielding one bug. Unlike compiler testing, PDF test cases require no inputs and no pre-processing of the training corpus. Skyfire [30] learns a probabilistic context-sensitive grammar over a corpus of programs to generate input seeds for mutation testing. The generated seeds are shown to improve the code coverage of AFL [33] when fuzzing XSLT and XML engines, though the seeds are not directly used as test cases. Machine learning has also been applied to other areas such as improving bug finding static analyzers [11, 15], repairing programs [17, 31], prioritizing test programs [5], identifying buffer overruns [8], and processing bug reports [14, 18]. To the best of our knowledge, no work so far has succeeded in finding compiler bugs by exploiting the learned syntax of mined source code for test case generation. Ours is the first to do so.

6 CONCLUSIONS

We present a novel framework for compiler fuzzing. By posing the generation of random programs as an unsupervised machine learning problem, we dramatically reduce the cost and human effort required to engineer a random program generator. Large parts of the stack are programming language-agnostic, requiring only a corpus of example programs, an encoder, and a test harness to target a new language.

We demonstrated our approach by targeting the challenging many-core domain of OpenCL. Our implementation, DeepSmith, has uncovered dozens of bugs in OpenCL implementations. We have exposed bugs in parts of the compiler where current approaches have not, for example in missing error handling. We provided a preliminary exploration of the extensibility of our approach. Our test cases are small, two orders of magnitude shorter than the state-of-the-art, and easily interpretable.

7 ACKNOWLEDGEMENTS

This work was supported by the UK EPSRC under grants EP/L01503X/1 (CDT in Pervasive Parallelism), and EP/P003915/1 (SUMMER).

REFERENCES

[1] O. Bastani, R. Sharma, A. Aiken, and P. Liang. Synthesizing Program Input Grammars. In *PLDI*, 2017.

[2] A. Betts, N. Chong, and A. Donaldson. GPUVerify: A Verifier for GPU Kernels. In *OOPSLA*. ACM, 2012.

[3] A. S. Boujarwah and K. Saleh. Compiler Test Case Generation Methods: A Survey and Assessment. *Information and Software Technology*, 39(9), 1997.

[4] S. R. Bowman, L. Vilnis, O. Vinyals, A. M. Dai, R. Jozefowicz, and S. Bengio. Generating Sentences from a Continuous Space. *arXiv:1511.06349*, 2015.

[5] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie. Learning to Prioritize Test Programs for Compiler Testing. In *ICSE*, 2017.

[6] 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.

[7] Y. Chen, A. Groce, C. Zhang, W. Wong, X. Fern, E. Eide, and J. Regehr. Taming Compiler Fuzzers. *PLDI*, 2013.

[8] M. Choi, S. Jeong, H. Oh, and J. Choo. End-to-End Prediction of Buffer Overruns from Raw Source Code via Neural Memory Networks. *arXiv:1703.02458*, 2017.

[9] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. End-to-end Deep Learning of Optimization Heuristics. In *PACT*. IEEE, 2017.

[10] P. Godefroid, H. Peleg, and R. Singh. Learn&Fuzz: Machine Learning for Input Fuzzing. In *ASE*, 2017.

[11] K. Heo, H. Oh, and K. Yi. Machine-Learning-Guided Selectively Unsound Static Analysis. In *ICSE*, 2017.

[12] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8), 1997.

[13] C. Holler, K. Herzig, and A. Zeller. Fuzzing with Code Fragments. *Usenix*, 2012.

[14] X. Huo, M. Li, and Z. Zhou. Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code. In *IJCAI*, 2016.

[15] U. Koc, P. Saadatpanah, J. S. Foster, and A. A. Porter. Learning a Classifier for False Positive Error Reports Emitted by Static Code Analysis Tools. In *MAPL*, 2017.

[16] A. S. Kossatchev and M. A. Posypkin. Survey of Compiler Testing Methods. *Programming and Computer Software*, 31(1), 2005.

[17] M. Koukoutos, M. Raghothaman, E. Kneuss, and V. Kuncak. On Repair with Probabilistic Attribute Grammars. *arXiv:1707.04148*, 2017.

[18] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports. In *ASE*, 2015.

[19] V. Le, M. Afshari, and Z. Su. Compiler Validation via Equivalence Modulo Inputs. In *PLDI*, 2014.

[20] C. Lidbury, A. Lascu, N. Chong, and A. Donaldson. Many-Core Compiler Fuzzing. In *PLDI*, 2015.

[21] Z. C. Lipton, J. Berkowitz, and C. Elkan. A Critical Review of Recurrent Neural Networks for Sequence Learning. *arXiv:1506.00019*, 2015.

[22] W. M. McKeeman. Differential Testing for Software. *DTJ*, 10(1), 1998.

[23] E. Nagai, A. Hashimoto, and N. Ishiura. Scaling up Size and Number of Expressions in Random Testing of Arithmetic Optimization of C Compilers. In *SASIMI*, 2013.

[24] R. Pacanu, T. Mikolov, and Y. Bengio. On the Difficulties of Training Recurrent Neural Networks. In *ICML*, 2013.

[25] M. Pflanzer, A. Donaldson, and A. Lascu. Automatic Test Case Reduction for OpenCL. In *IWOCL*, 2016.

[26] J. Price and S. Mcintosh-Smith. Oclgrind: An Extensible OpenCL Device Simulator. In *IWOCL*. ACM, 2015.

[27] A. Radford, R. Jozefowicz, and I. Sutskever. Learning to Generate Reviews and Discovering Sentiment. *arXiv:1704.01444*, 2017.

[28] M. Raghu, B. Poole, J. Kleinberg, S. Ganguli, and J. Sohl-Dickstein. On the expressive power of deep neural networks. *arXiv:1606.05336*, 2016.

[29] C. Sun, V. Le, and Z. Su. Finding Compiler Bugs via Live Code Mutation. In *OOPSLA*, 2016.

[30] J. Wang, B. Chen, L. Wei, and Y. Liu. Skyfire: Data-Driven Seed Generation for Fuzzing. In *S&P*, 2017.

[31] M. White, M. Tufano, M. Martínez, M. Monperrus, and D. Poshyvanyk. Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. *arXiv:1707.04742*.

[32] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and Understanding Bugs in C Compilers. In *PLDI*, 2011.

[33] M. Zalewski. American Fuzzy Lop.

[34] Q. Zhang, C. Sun, and Z. Su. Skeletal Program Enumeration for Rigorous Compiler Testing. In *PLDI*, 2017.