DeepSmith: Compiler Fuzzing through Deep Learning

Chris Cummins[*,1] Pavlos Petoumenos[*],
Alastair Murray[†], Hugh Leather[*]

[*]*University of Edinburgh, Scotland, UK*
[†]*Codeplay Software, Edinburgh, Scotland, UK*

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 in OpenCL compilers 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 $3.03\times$ 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.

KEYWORDS: Deep Learning; Compilers; Differential Testing; OpenCL

1 Introduction

Compilers should produce correct code for valid inputs, and meaningful errors for invalid inputs. Properly testing compilers is hard — modern optimizing compilers are large and complex programs, and their input space is huge. Random test case generation — *fuzzing* — is a well established and effective method for identifying compiler bugs. 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. Differential Testing can be used to find compiler bugs in generated binaries without the need for

[1]E-mail: c.cummins@ed.ac.uk
[2]This work was supported by the UK EPSRC under grants EP/L01503X/1 (CDT in Pervasive Parallelism), and EP/P003915/1 (SUMMER).

an oracle by comparing program outputs across compilers. 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.

CSmith, a state-of-the-art fuzzer which randomly enumerates programs from a subset of the C programming language grammar, has been successfully used to identify hundreds of bugs in C compilers, but developing such a fuzzer is a huge undertaking. 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 [3] — 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.

We propose 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. 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. 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[3] is our open source framework for compiler fuzzing. Figure 1 provides a high-level overview. The three key components of DeepSmith are: a generative model for random programs, a test harness, and voting heuristics for differential testing.

Generative Model We treat the generation of compiler testing inputs as an unsupervised machine learning task, employing state-of-the-art deep learning techniques to build models for how humans write programs. To generate inputs for testing OpenCL compilers, we train a Recurrent Neural Network on a corpus of handwritten code, assembled by mining 10k OpenCL kernels from open source repositories on GitHub [2]. We used an *oracle compiler*

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

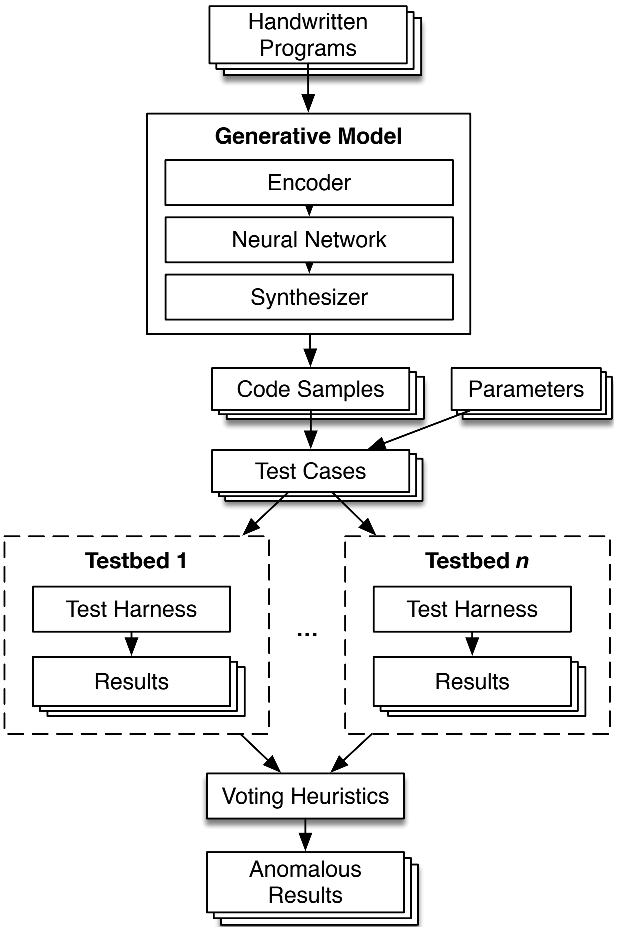

Figure 1: DeepSmith system overview.

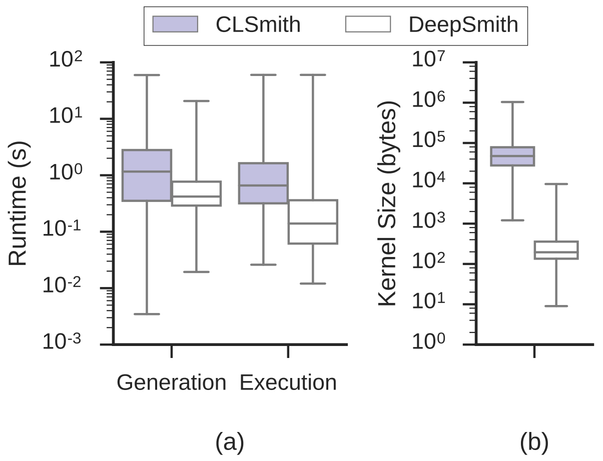

Figure 2: Comparison of runtimes (a) and test case sizes (b) of DeepSmith and CLSmith [3].

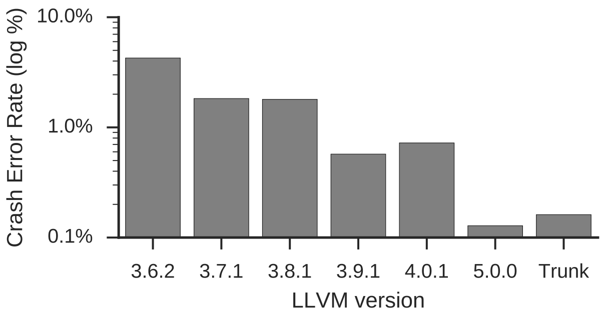

Figure 3: Crash rate of the Clang front-end of every LLVM release in the past 24 months compiling 75k DeepSmith kernels.

(LLVM 3.9) to statically check the source files, discarding files that are not well-formed. This corpus, exceeding one million lines of code, is encoded using a hybrid keyword and character level tokenizer [1]. 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. The trained network is sampled to generate new programs. To generate a new program, the model is seeded with the start of a program, 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 program can be detected and sampling halted.

Test Harness We developed a harness to execute generated OpenCL programs. The harness accepts an arbitrary OpenCL program, generates input data for it, and executes the program on an OpenCL device. 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.

Voting Heuristics We employ established Differential Testing methodologies to expose compiler defects. 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. A series of heuristics detect common causes of undefined behavior and remove false positives.

3 Evaluation

We compare our fuzzer to CLSmith [3], the state-of-the art OpenCL fuzzer. We conducted 2000 hours of automated testing across 10 OpenCL compilers (3 GPUs, 4 CPUs, a co-processor, and an emulator). DeepSmith 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 found that DeepSmith is able to identify a broad range of defects, many of which CLSmith cannot. For example, a common pattern in OpenCL programs is to obtain the thread identity and 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 pattern). DeepSmith, having modeled the frequency with which this occurs in real handwritten code, generates many permutations of this pattern. And in doing so, exposed a bug in the optimizer of two Intel compilers which causes the `if` branch of a DeepSmith-generated program to be erroneously executed when the kernel is compiled with optimizations enabled. CLSmith does not permit the thread identity to modify control flow, rendering such productions impossible.

Figure 2 compares the runtime and program sizes of the two approaches. DeepSmith test cases are on average evaluated $3.03\times$ faster than CLSmith ($2.45\times$, and $4.46\times$ for generation and execution, respectively), and are two orders of magnitude smaller.

The Clang front-end to LLVM is commonly used in OpenCL drivers. This in turn causes Clang-related defects to potentially affect multiple compilers. To evaluate the impact of Clang, we used debug+assert builds of every LLVM release in the past 24 months to compile 75k DeepSmith test cases. Figure 3 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. Notably, the current development trunk has the second lowest crash rate, emphasizing that compiler validation is a moving target. Since LLVM will not release unless their compiler passes their own extensive test suites, this also reinforces the case for compiler fuzzing.

4 Conclusions

We present a novel tool 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. Our implementation, DeepSmith, has uncovered dozens of bugs in OpenCL implementations, including in parts of the compiler where current approaches are not able to. Our test cases are small, two orders of magnitude shorter than the state-of-the-art, and easily interpretable.

References

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

[2] C. Cummins, P. Petoumenos, W. Zang, and H. Leather. Synthesizing Benchmarks for Predictive Modeling . In *CGO*. IEEE, 2017.

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