On the Naturalness of Fuzzer Generated Code

Anonymous Author(s)

ABSTRACT

Compiler fuzzing tools such as Csmith have uncovered a number of real-world bugs by randomly sampling programs using a generative model. The success of fuzzing is often attributed to its ability to generate unexpected corner case inputs that are easy to overlook during manual testing. Due to their chaotic nature, fuzzer-generated test cases can be notoriously hard to read, leading to the reliance on minimization tools such as C-Reduce (for C compiler bugs) which reduce the size of fuzzer-generated counterpart programs.

Researchers have also shown that human-written software has properties similar to natural language, in that it tends to be highly repetitive and predictable. Recent work has also indicated that humans prefer more predictable code, and that buggy code is often relatively unnatural.

In this paper, we study similar properties of naturalness on, perhaps unintuitively, fuzzer-generated code. In particular, we investigate whether fuzzer-generated code is unpredictable relative to human-written code, whether fuzzer-found bugs are even more unpredictable, whether minimization tools improve predictability, and if human-written code is unpredictable under constraints of the language structure of fuzzer-generated code.

Our findings are surprising: Csmith fuzzer-generated programs are *more* predictable than human-written C programs, and the minimization tool C-Reduce does not monotonically increase predictability during its reduction process. Our initial results suggest potential research directions in incorporating predictability metrics in the fuzzing and reduction tools themselves.

Index Terms—entropy, predictability, generation-based fuzzing, neural language modeling

ACM Reference Format:
Anonymous Author(s). 2021. On the Naturalness of Fuzzer Generated Code. In *Proceedings of The 44th International Conference on Software Engineering (ICSE 2022).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Fuzz testing is an effective method for automatic software testing, commonly used to find bugs in tools such as compilers, parsers, interpreters, and web browsers. These tools, often just called *compiler fuzzers*, generate vast numbers of programs using randomized algorithms often triggering real bugs in the software that processes such programs [6, 8, 9, 11, 19]. However, due to the chaotic nature of the random input generation process, the resulting bug-triggering programs are often difficult to interpret. The process of test-input minimization [16, 20], which reduces these inputs to much smaller programs while still triggering the original bug, has been proposed to aid in the interpretability of fuzzer-generated programs. The reduced programs are often orders of magnitude smaller, highlighting that the vast majority of fuzzer-generated code is not relevant for triggering the found bug.

In a thus far unrelated area of research, researchers have discovered that human-written source code is surprisingly predictable and repetitive, similar to natural languages [7], and amenable to modeling using the same type of language models. Such models have since demonstrated that buggy code is relatively less predictable [12], and that predictability tends to correlate with readability [4]. The confluence of these two lines of work suggests the natural question: are fuzzers effective at triggering bugs because their inputs are highly unpredictable? And does reducing such bug-triggering programs yield more predictable (and thus perhaps more readable) programs?

In this paper, we present the first empirical study of the entropy, or "naturalness" of *fuzzer-generated code*. Our initial hypothesis is that fuzzer-generated programs would have high entropy due to the random input generation process. We also study the entropy of fuzzer-generated programs that have revealed compiler bugs. Further, we investigate whether test-case reduction improves predictability, assuming a strong connection to readability. We conduct this study on C programs, using Csmith [19] and C-Reduce [16] as a model generation-based compiler fuzzer and test-case reducer respectively. We report surprising results:

(1) Fuzzer-generated C programs are on average *more repetitive* than human written code, possibly suggesting that these tools are not exploring the most abnormal programs.
(2) Reported Fuzzer-found bugs in C compilers were in turn more predictable than randomly generated programs.
(3) The C-Reduce test case reduction tool improves code predictability on average, but only slightly and inconsistently.

As such, our findings have ramifications for generation-based fuzz testing, test case minimization tools, predictability metrics, and statistical language modeling. In addition, we lay out how future research in multiple fields could incorporate the repetitive nature of automatically generated test cases. This future research could lead to methods that incorporate predictability metrics in test case generation to either generate chaotic programs or programs that are not typically created by developers.

2 BACKGROUND

While code readability is subjective and requires human reviewers to measure, user studies have shown it to be connected to predictability according to well-equipped language models [4]. Our empirical study builds on previous work that highlights the reduced predictability of buggy code [12], as well as on the observation that fuzzer-generated programs appear particularly chaotic to a human reader, which is often conjectured to be the reason behind their bug-revealing capability. This empirical study combines these two

ICSE 2022, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

observations into the conjecture that fuzzer-generated code ought to be highly unpredictable to a language model trained on human-written code. To measure predictability, we use a neural language model trained on publicly available C code.

2.1 Compiler Fuzzing with CSmith

Generation-based fuzzing is a methodology for finding software bugs by randomly sampling program inputs using a model of input format. The program under test is executed with each sampled input; a program crash or other unexpected behavior reveals a bug. In this paper, we focus on Csmith [19], a generation-based fuzz testing tool for finding bugs in C compilers. Csmith specializes in randomly sampling C programs that have well defined behavior as per the C99 standard. It identifies compiler bugs by checking if the generated C program produces the same result when compiled by various C compilers such as GCC, LLVM/Clang, Intel CC, etc. If these compilers produce different results (a *miscompilation* bug), or if that compiler crashes while performing compilation (a *crash compile* bug), Csmith has revealed a C compiler bug. Csmith has been successful in discovering dozens of previously unknown bugs in compilers such as GCC [2] and LLVM/Clang [3]. The success of compiler fuzzing is sometimes attributed to its ability to generate artificial programs that are unlikely to appear *in the wild* [10]; that is, the bug-trigger programs are corner cases that do not fit the pattern of common human-written C programs.

2.2 Test-Case Minimization with C-Reduce

Fuzzer-generated programs typically grow very large before they trigger bugs. The resulting programs can often be hundreds of lines of code, even though most of the code is not responsible for triggering the compiler bug. It is naturally hard for developers to identify the bug-triggering root cause in such large and complex programs. To aid readability and debugging, minimization tools such as C-Reduce [16] have been developed. C-Reduce is an automated minimization tool for C/C++ programs. It implements a greedy algorithm, similar to delta debugging [20], which iteratively removes code fragments that do not affect the existence of a given property of that file (e.g., triggering a compiler crash). Since C-Reduce helps in improving the debuggability of bug-triggering programs, we also expect the results of minimization to be more readable, and correspondingly, more "natural" when compared to human-written programs.

2.3 Language Models of Code

Statistical language models learn generative probabilities of text. Highly predictable code yields low *entropy* values, a metric tied to the average probability of a phrase, and vice versa. We use the term cross-entropy when the model is trained on a distinct corpus (body of text) from the one it provides predictability scores for [5]. In this setting, cross-entropy captures the degree of information transferred from the training corpus to the tested one.

Empirical studies have shown that predictability, as computed by language models of code, tends to correlate with source code readability [4]. Language models trained on software artifacts have been used for applications such as code completion [14], idiom mining [1], and de-obfuscation [13]. Studies have also shown that buggy code has relatively higher entropy compared to non-buggy code [12]. However, to the best of our knowledge, statistical language models have not been previously employed on fuzzer-generated programs.

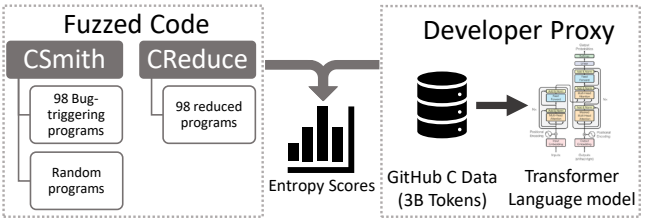

Figure 1: Outline of the methodology. We investigated the entropy of two corpora: the Csmith fuzzer-generated programs and C-Reduced minimized programs.

3 MOTIVATION AND METHODOLOGY

Our research is thus motivated by three main questions: (1) Is fuzzer-generated code really unpredictable compared to human-written programs? (2) Do fuzzer-generated bugs possess higher cross-entropy? (3) Does test-case minimization reduce cross-entropy?

Figure 1 shows our approach: we focus on the Csmith compiler fuzzer, collecting both a modest corpus of 98 reported bug-triggering programs and a much larger corpus of randomly generated programs. We contrast the former with their C-Reduce'd counterparts.[1] We then collected a large corpus of developer-written C/C++ programs from GitHub, on which we train a Transformer language model [18]. Once trained, we compute the entropy of our test corpora: the bug-triggering Csmith programs, their C-Reduce'd counterparts, randomly generated Csmith programs, and a test set of human-written C programs.

3.1 Csmith-Generated Program Corpus

We first collected a corpus of 98 compiler-bug-triggering programs that were known to be generated by Csmith. These programs come from a GitHub repository [15] and the evaluation dataset from the PLDI'12 paper by Regehr et al. [16]. For each of these C programs, we also obtained corresponding minimized test cases after processing with C-Reduce. Thus, we have an additional 98 auto-generated-and-minimized C programs. Finally, we used Csmith with its default configuration to randomly sample 1,000 additional C programs; we do not expect these programs to trigger any compiler bugs. We refer to the programs in this corpus as "random" or "non-buggy" Csmith programs.

3.2 Training Corpora

In order to learn a model of human-written code, we collected a corpus of C programs from GitHub. We mined the 1,217 most starred GitHub repositories that primarily use the C programming language. This amounted to 745K C, 56K C++, and 602K H/H++ (header) files. We used Pygments[2] to tokenize these files, discarding comments, which do not appear in automatically generated code. This produced a corpus of slightly over 3 billion tokens. We next

[1] Such reduction is not an option for the randomly generated programs because they obey no obvious property for the reduction process to preserve.
[2] https://pygments.org/

Table 1: Entropy results on each test corpus under both a model trained on developer data (H_{human}) and on Csmith generated files (H_{Csmith})

	Files	Tokens/file	H_{human}	H_{Csmith}
GitHub	52,041	2,794	5.63	8.61
Csmith Random	1,000	38,431	5.20	1.35
Csmith Bugs	98	23,686	4.88	2.37
C-Reduce'd Bugs	98	76	4.66	4.72

used the SentencePiece encoder[3] to construct a sub-token vocabulary of the 25,000 most common tokens using byte-pair encoding, and encoded our inputs accordingly.

3.3 Model Training

We trained a standard size Transformer model [18], with 6 layers and 512-dimensional attention across 8 attention heads. Our language model processes entire files in chunks of 512 tokens (roughly 20-40 lines) due to memory limitations of attention. We processed these in batches of 256 chunks across four RTX8000 GPUs. The model took about one week to converge. We split the collected repositories by their containing organization into a training data set, validation data set, and test data set on a 90%/5%/5% distribution spread. A separate Transformer with the same architecture was trained on a randomly generated Csmith corpus of the same size.

3.4 Metrics

To judge the predictability of C/C++ files in our work, we compute the per-token average entropy [5] for each train/test pair. We investigate the contrast between the GitHub model's entropy, which is presumably "natural" to developers, relative to both other GitHub data and (cross-entropy) our Csmith/C-Reduce corpora. After entropy scoring each file within a corpus given a specific trained model, the average entropy of the entire corpus can be calculated using either simple averaging of file-level scores or weighting these by file length. We opted for the former, but found the results to differ little.

4 RESULTS

Our primary result concerns the relative predictability of fuzzer-generated programs given a language model trained on human code. Table 1 shows summary stats of our evaluation corpora and corresponding entropy values. For now, the rightmost column can be ignored. The second-to-last column in Table 1, labelled H_{human}, shows a comparison of the average per-token entropy of each test corpus. Most surprisingly, the developer corpus is *the least* predictable one here, having the largest entropy value of 5.63. Randomly generated Csmith programs have an average entropy of 5.20, which is lower than the human written code corpus by a small margin,[4] *despite* the model being trained on a corpus resembling the human-produced programs, not Csmith ones! In other words, even to a model based on human-produced code, ***automatically generated C test cases are more repetitive than human-written code.***

[3]https://github.com/google/sentencepiece

[4]The difference is about 0.43 base-e bits (called 'nats'), which corresponds to about a 1.5x difference in (geometric) mean predictability.

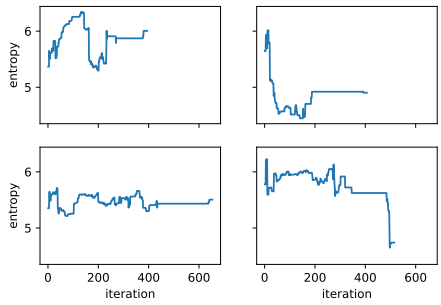

Figure 2: Average entropy of several bug-triggering programs as they get minimized by C-Reduce [16].

The next two rows in Table 1 show results for bug-triggering programs. As explained in Section 3.1, these appear in two forms: the originally generated C file that first exposed a new compiler bug, and the smallest reduced snippet of code that triggers the same bug. We find these programs to be more predictable still, having entropy of 4.88 initially and 4.66 when reduced. This is surprising. In regular source code, buggy programs tend to be more entropic than bug-free code, which echos the intuition that bugs tend to appear in code that is more complex and harder to read [12]. This is not the case here. In fact, the reduced programs are a full (base-e) bit lower—and substantially smaller—than human-written programs.

4.1 From the Fuzzer's Perspective

Our results so far have focused on cross-entropy measures based on a language model of human-written code. To improve our understanding, we also decided to train a language model that learns the *distribution of fuzzer-generated code*. The final column in Table 1, labelled H_{Csmith}, shows the same evaluations as before, but from the perspective of a model trained on a similar volume of Csmith code, using the same configuration and resources. The contrast is stark. This model finds developer-written code far more surprising than vice versa. It is also *far* more accurate at predicting new Csmith programs. This partly explains why the human-based model found Csmith programs so predictable: they are *inherently* tremendously repetitive. The human-written code model only scratched the surface of how much.

Here, the bugs do follow a more expected trend. The original (unreduced) bug-triggering programs are nearly twice as entropic (and thus unpredictable) as the randomly generated programs. This suggests that such programs are still relatively rare and complex compared to randomly sampling valid programs from the C99 grammar, which may explain why most Csmith programs do not trigger bugs. Once reduced, this entropy climbs significantly further, though still well below that of human-written programs, presumably due to the elimination of many repetitive parts of the program. This final entropy is most similar between the two models, perhaps reflecting that these programs are not particularly representative of either corpus, but simple enough to be recognizable to both.

4.2 Entropy Changes During Reduction

Table 1 indicated that the test case reduction tool C-Reduce tends to decrease the (GitHub-based) entropy of bug triggering programs.

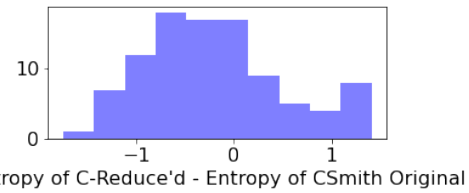

Figure 3: Entropy differences between original and reduced files, according to a model trained on human-written code.

Figure 3 summarizes this trend more specifically, in terms of the distribution of entropy changes. This reflects a substantial spread.

Because C-Reduce operates by steadily deleting snippets of code, we can also compute the cross-entropy of intermediate solutions during its minimization process. We wondered whether entropy decreased monotonically as C-Reduce chops away redundant code, assuming that each intermediate step is more debuggable than the previous one. Figure 2 plots the entropy during this process for four buggy programs; the horizontal axis corresponds to iterations in the reduction algorithm. Overall, we found no consistent trend; while C-Reduce tends to reduce the average entropy, individual reduction steps rarely have a consistent effect either way, and entropy mostly changes in large jumps when a significant chunk of code is removed (often, but not always early on). Intuitively, if a reduction increases average entropy, then the snippet of code that was removed had a lower average entropy than the rest of the file. In the case of test case reduction, it is important to keep in mind that low average entropy alone is not a goal: highly predictable but irrelevant code should always be removed.

5 IMPLICATIONS

Our finding that fuzzer-generated C files are more predictable to a language model than human-written code has potentially significant ramifications for future research, which we discuss here.

For Bug-Detection:
A key aspect of our motivation for studying this intersection of methods was the established finding that buggy code is more entropic [12]. We expected this to hold even more for fuzzer-generated programs, but to the contrary, the latter were more predictable both with and without reduction. This echoes the finding that supports CReduce itself: while fuzzers discover bugs through large-scale chaotic generation, the actual bug-triggering parts involve small and mostly simple code. Our results add to this by showing that these programs are also overwhelmingly predictable on average.

This raises two natural questions: why did humans not trigger these bugs seen in ostensibly “natural" programs; and, why does Csmith not constantly produce bug-triggering examples, given that those appear to be at least as simple as its typical programs? The latter finding might lead one to conclude that entropy and bugginess do not correlate for fuzzer-generated code, but our analysis involving a model trained on Csmith code shows quite the opposite: bug-triggering programs are **much less predictable** from Csmith’s perspective. Evidently, discovering bugs requires it to generate programs that are about twice as improbable (across 10Ks of tokens) as its typical programs. The answer to the first question above is not as obvious. From anecdotal inspection, these bugs often contain at least one highly unusual construct, plus a substantial amount of fairly repetitive boilerplate. Perhaps triggering bugs primarily requires the presence of statements with a high “peak entropy". We believe that this is worth investigating further.

For the Future of Fuzzers:
Our exploratory study opens a new research direction at the intersection of fuzzers and language models. In particular, generating valid C programs, as Csmith does, requires satisfying a large number of constraints, which likely contributes to generating very repetitive program. Other fuzzers may exhibit different behavior: some leverage a similar AST-expansion approach but involve far fewer constraints, while others use a different approach entirely—e.g., mutation-based fuzzers, protocol-based fuzzers. To draw broader conclusions about the naturalness of fuzzer-generated code requires a more comprehensive evaluation. Our hypothesis is that the high degree of repetition seen in Csmith code is an artifact of both the use of hand-crafted rules (e.g., we often saw specific integers, like 128), and of the AST-driven approach to growing programs. The latter involves expanding randomly selected nodes, which somewhat paradoxically tends to result in *more* repetitive programs in our experience, for instance, by generating very long if-conditions with near-identical clauses.

Given the high repetition in automatically generated programs, including bug-triggering ones, it seems more than likely that a very large portion of the program space has yet to be explored. While we cannot guarantee that discovering less predictable test cases will lead to finding more bugs, our results clearly indicate that bugs may reside there. This strongly suggests that a future for generation-based fuzzing tools could be to incorporate language modeling to create test cases that are *highly unpredictable by design*, rather than just random sampling from a predictible distribution.

Our study of predictability of C-Reduce’d program also has ramifications for test-case minimization. If the goal of post-processing is to improve human consumption of buggy test inputs, can they be improved by taking cross-entropy into account? Work such as DeepTC-Enhancer [17] has highlighted the importance of improving the readability of auto-generated tests, and we believe this direction is worth pursuing further.

6 CONCLUSION

An analysis using a large language model strongly indicates that (compiler) fuzzer-generated test cases, including bug-triggering ones and their reduced counterparts, are more repetitive than human-written code. This holds both from the perspective of a model trained exclusively on “natural" developer-written programs, and (far more so) from a model trained on fuzzer-generated programs. The latter evidences that these programs are in fact tremendously repetitive and predictable, defying the conventional wisdom that fuzzers emit highly chaotic code and thereby expose new bugs. Instead, bug-triggering programs (reduced or not) are more predictable than the average C program, although they do appear surprising to a model trained on random, non-bug triggering code. This finding informs our vision of guiding fuzzers directly with entropy, and highlights the need for similar investigations of other types of fuzzers as well as test-case minimization tools.

REFERENCES

[1] Miltiadis Allamanis and Charles Sutton. 2014. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. 472–483.

[2] CSmith authors. 2011. Csmith - GCC Bugs Reported. https://embed.cs.utah.edu/csmith/gcc-bugs.html Retreived October 14, 2021.

[3] CSmith authors. 2011. Csmith - GCC Bugs Reported. https://embed.cs.utah.edu/csmith/llvm-bugs.html Retreived October 14, 2021.

[4] Casey Casalnuovo, Kevin Lee, Hulin Wang, Prem Devanbu, and Emily Morgan. 2020. Do Programmers Prefer Predictable Expressions in Code? *Cognitive Science* 44, 12 (2020), e12921.

[5] Stanley F Chen and Joshua Goodman. 1999. An empirical study of smoothing techniques for language modeling. *Computer Speech & Language* 13, 4 (1999), 359–394.

[6] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 197–208.

[7] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

[8] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security '12)*. 445–458.

[9] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226.

[10] Michaël Marcozzi, Qiyi Tang, Alastair F Donaldson, and Cristian Cadar. 2019. Compiler fuzzing: How much does it matter? *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[11] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing javascript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1629–1642.

[12] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "naturalness" of buggy code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 428–439.

[13] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from" Big Code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)*. 111–124.

[14] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. 419–428.

[15] John Regehr. 2016. Artisinal Compiler Crashes. https://github.com/regehr/compiler-crashes Retreived October 14, 2021.

[16] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 335–346.

[17] Devjeet Roy, Ziyi Zhang, Maggie Ma, Venera Arnaoudova, Annibale Panichella, Sebastiano Panichella, Danielle Gonzalez, and Mehdi Mirakhorli. 2020. DeepTC-Enhancer: Improving the Readability of Automatically Generated Tests. In *IEEE/ACM International Conference on Automated Software Engineering (ASE'20)*. IEEE.

[18] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[19] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.

[20] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.