On the Naturalness of Fuzzer-Generated Code

Rajeswari Hita Kambhamettu*
rkambham@cmu.edu
Carnegie Mellon University
Pennsylvania, USA

John Billos*
billjr20@wfu.edu
Wake Forest University
North Carolina, USA

Tomi Oluwaseun-Apo*
cfo5115@psu.edu
Pennsylvania State University
Pennsylvania, USA

Benjamin Gafford
bgafford@cmu.edu
Carnegie Mellon University
Pennsylvania, USA

Rohan Padhye
rohanpadhye@cmu.edu
Carnegie Mellon University
Pennsylvania, USA

Vincent J. Hellendoorn
vhellendoorn@cmu.edu
Carnegie Mellon University
Pennsylvania, USA

ABSTRACT

Compiler fuzzing tools such as Csmith have uncovered many bugs in compilers by randomly sampling programs from a generative model. The success of these tools is often attributed to their ability to generate unexpected corner case inputs that developers tend to overlook during manual testing. At the same time, their chaotic nature makes fuzzer-generated test cases notoriously hard to interpret, which has lead to the creation of input simplification tools such as C-Reduce (for C compiler bugs). In until now unrelated work, researchers have also shown that human-written software tends to be rather repetitive and predictable to language models. Studies show that developers deliberately write more predictable code, whereas code with bugs is relatively unpredictable. In this study, we ask the natural questions of whether this high predictability property of code also, and perhaps counter-intuitively, applies to fuzzer-generated code. That is, we investigate whether fuzzer-generated compiler inputs are deemed unpredictable by a language model built on human-written code and surprisingly conclude that it is not. To the contrary, Csmith fuzzer-generated programs are *more* predictable on a per-token basis than human-written C programs. Furthermore, bug-triggering tended to be more predictable still than random inputs, and the C-Reduce minimization tool did not substantially increase this predictability. Rather, we find that bug-triggering inputs are unpredictable relative to *Csmith's own* generative model. This is encouraging; our results suggest promising research directions on incorporating predictability metrics in the fuzzing and reduction tools themselves.

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

ACM Reference Format:
Rajeswari Hita Kambhamettu, John Billos, Tomi Oluwaseun-Apo, Benjamin Gafford, Rohan Padhye, and Vincent J. Hellendoorn. 2022. On the Naturalness of Fuzzer-Generated Code. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3524842.3527972

*These authors contributed equally to the paper.

MSR '22, May 23–24, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9303-4/22/05.
https://doi.org/10.1145/3524842.3527972

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 [7, 9, 10, 12, 19]. However, due to the random nature in the input generation process, the resulting bug-triggering programs are often difficult to interpret. As a response, test-input minimization techniques have been proposed to aid in the interpretability of fuzzer-generated programs [17, 20], which reduce these inputs to much smaller programs while still triggering the original bug. 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 to models designed to capture repetition in natural languages [8]. As a consequence, this high predictability of code is often referred to as "naturalness". Language models have since demonstrated that buggy code is relatively less predictable than correct code [13], and that predictability tends to correlate with readability [5]. The confluence of studies of fuzzing with this line of work suggests using these same models to answer the natural question: are fuzzers effective at triggering bugs because their inputs are highly unpredictable (hence why programmers might miss these in their test cases)? 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 "naturalness" of *fuzzer-generated code*, specifically in the context of compiler bug triggering C/C++ programs generated with Csmith [19]. Our initial hypothesis is that fuzzer-generated programs ought to be unpredictable, due to their random input generation process and anecdotally chaotic outputs. Paradoxically, we find these programs to be rather predictable – more so than C code written by real developers. Yet these fuzzer-generated programs are from what anyone would call natural! More surprisingly, bug-triggering inputs were not any less predictable, nor did input reduction using C-Reduce [17] improve code predictability consistently or substantially.

In response to these counter-intuitive findings, we conjecture that Csmith produces remarkably repetitive programs by nature and train a new "language" model entirely on fuzzer-generated programs. This model predicts new Csmith programs with high accuracy and discriminated much more clearly between bug-triggering

and ineffective inputs, with the former being substantially less predictable to this model. Our findings suggest that fuzzers (at least Csmith) produce highly repetitive programs and that bug-triggering inputs tend to emerge in programs that these tools are unlikely to generate, which explains why many generated programs do not trigger bugs. These results may inform both generation-based fuzz testing and test case minimization tool design, the former of which may want to explicitly optimize for unpredictable programs, and the latter for the reverse.

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 [5]. Our empirical study builds on previous work that highlights that buggy code is less predictable [13], as well as on the observation that fuzzer-generated programs tend to 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 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 [3] and LLVM/Clang [4]. The success of compiler fuzzing is sometimes attributed to its ability to generate artificial programs that are unlikely to appear *in the wild* [11]; 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. These programs become burdensome for developers as identifying the bug-triggering root because of their large-scale and complexity. To make these inputs easier to debug, minimization tools such as C-Reduce [17] 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 similar to human-written programs.

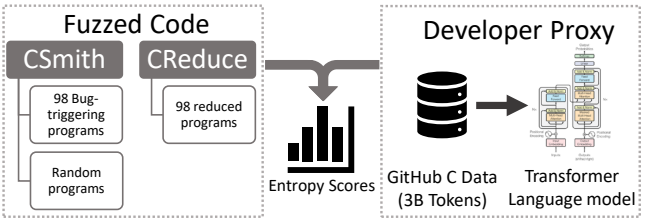

Figure 1: Outline of the methodology. We investigated the entropy of two corpora: the Csmith fuzzer-generated programs and C-Reduced minimized programs.

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. 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 [6]. In this setting, cross-entropy captures the amount of information transferred from the training corpus to the tested one.

Our evaluation focuses on predictability as computed by language models. Empirical studies have shown that this predictability *tends to* correlate with source code readability in certain settings [5], but it is not a given that more predictable code is more readable. Similarly, the high degree of predictability that is typical of code under such models has often been referred to as its *naturalness* [8], but naturalness does not have a precise definition based on predictability – indeed, this work shows highly predictable code that is decidedly unnatural and unreadable. Language models trained on software artifacts have been used for applications such as code completion [15], idiom mining [2], and de-obfuscation [14]. Studies have also shown that buggy code has relatively higher entropy compared to non-buggy code [13]. However, to the best of our knowledge, statistical language models have not been previously employed on fuzzer-generated 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 (cross-)entropy

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

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 [16] and the evaluation dataset from the PLDI'12 paper by Regehr et al. [17]. 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 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. We chose to use Transformers over perhaps N-gram based models because they consider substantially longer contexts and are therefore more accurate at program prediction. 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. The test corpus was de-duplicated both internally and against the training corpus at the file level [1]. 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 [6] for each train/test pair. We investigate the contrast between the GitHub model's entropy, which is presumably "natural" to developers [8], 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.

[2]https://pygments.org/
[3]https://github.com/google/sentencepiece

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	2,086	6.02	9.38
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

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 average entropy value of 6.02. 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 statistically predictable than human-written code.*** We reason that while CSmith code might appear to be easier to reason about, the auto-generated variable names are generally highly unusual. This surprising result is likely because of the high degree of repetition present in CSmith code.

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 entropic 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 predictable than bug-free code, which echos the intuition that bugs tend to appear in code that is more complex and harder to read [13]. 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

[4]The difference is ca. 0.82 base-e bits (called 'nats'), which corresponds to about a 2.3x difference in (geometric) mean predictability.

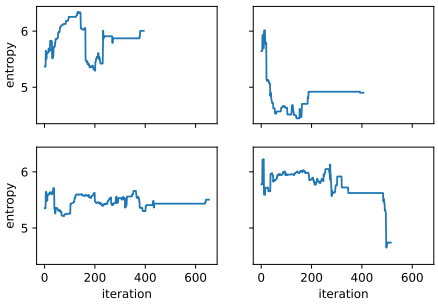

Figure 2: Average entropy of several bug-triggering programs as they get minimized by C-Reduce [17].

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, presumably because it was looking for developer-like repetitive patterns.

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, though still well below that of human-written programs, likely due to the elimination of many highly predictable 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 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). 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.

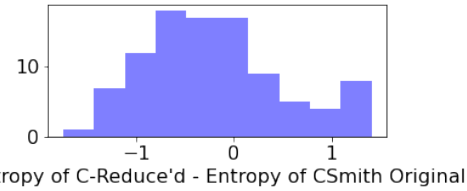

Figure 3: Entropy differences between original and reduced files, according to a model trained on human-written code.

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 [13]. 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 random generation that tends to be chaotic, the actual bug-triggering parts are often very small. For the Csmith fuzzer, fuzzer-generated programs are clearly highly repetitive, which perhaps explains why they can be reduced so easily.

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 supports a more nuanced conclusion: bug-triggering programs are **much less predictable** from this fuzzer'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:
Generating valid C programs as Csmith does requires satisfying a large number of constraints. This likely contributes to generating very repetitive program, as does the frequent application of hand-crafted rules (e.g., we often saw specific integers, like 128). Generating random ASTs in this way may somewhat paradoxically result in *more* repetitive programs; for instance, we often saw this lead to very long if-conditions with near-identical clauses. Our exploratory study opens a new research direction at the intersection of fuzzers and language models. 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. It is not yet clear how this would affect their outputs' predictability.

Given the lack of variety in automatically generated program statements, 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 random sampling from a predictable distribution.

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, produce programs that are more predictable than human-written code. This holds both from the perspective of a model trained exclusively on "natural" developer-written programs, thereby challenging the equivocation of "naturalness" with highly predictable programs, and (far more so) from a model trained on fuzzer-generated programs. The latter evidences that these programs are in fact tremendously repetitive, defying conjecture that fuzzers expose new bugs through emitting highly chaotic code. Instead, bug-triggering programs (reduced or not) are more predictable than the average C program, although they do appear relatively surprising to a model trained on random, non-bug triggering fuzzer-generated 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.

7 ACKNOWLEDGEMENTS

This research was funded in part by NSF grants CCF-1852260 and CCF-2120955, and partially by JP Morgan. We thank the CSmith and C-Reduce teams for their open source tools. In particular, we thank Eric Eide for sending our team copies of 98 fuzzer-found bugs and corresponding C-Reduced files, which we used in this study.

REFERENCES

[1] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 143–153.

[2] 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.

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

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

[5] 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.

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

[7] 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.

[8] 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.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[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.