SoftwareX 15 (2021) 100711

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

Cnerator: A Python application for the controlled stochastic generation of standard C source code

Francisco Ortin [a,b,*], Javier Escalada [a]

[a] University of Oviedo, Computer Science Department, Federico Garcia Lorca 18, 33007, Oviedo, Spain
[b] Munster Technological University, Department of Computer Science, Rossa Avenue, Bishopstown, Cork, Ireland

ARTICLE INFO

Article history:
Received 25 January 2021
Received in revised form 7 May 2021
Accepted 13 May 2021

Keywords:
Big code
Mining software repositories
Machine learning
C programming language
Stochastic program generation
Python

ABSTRACT

The Big Code and Mining Software Repositories research lines analyze large amounts of source code to improve software engineering practices. Massive codebases are used to train machine learning models aimed at improving the software development process. One example is decompilation, where C code and its compiled binaries can be used to train machine learning models to improve decompilation. However, obtaining massive codebases of portable C code is not an easy task, since most applications use particular libraries, operating systems, or language extensions. In this paper, we present Cnerator, a Python application that provides the stochastic generation of large amounts of standard C code. It is highly configurable, allowing the user to specify the probability distributions of each language construct, properties of the generated code, and post-processing modifications of the output programs. Cnerator has been successfully used to generate code that, utilized to train machine learning models, has improved the performance of existing decompilers. It has also been used in the implementation of an infrastructure for the automatic extraction of code patterns.

Code metadata

Current code version	v.1.0.1
Permanent link for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-21-00022
Legal code license	BSD 3-Clause
Code versioning system used	Git
Software code language used	Python 3.7+
Compilation dependencies	numpy PyPI Python package
Developer documentation	https://computationalreflection.github.io/Cnerator
Support email for questions	ortin@uniovi.es

Software metadata

Current software version	v.1.0.1
Permanent link to executables of this version	https://github.com/ComputationalReflection/Cnerator/releases
Legal software license	BSD 3-Clause
Operating system	Linux, Windows and Mac OS
Installation requirements & dependencies	numpy PyPI Python package
User manual	https://github.com/ComputationalReflection/Cnerator/blob/main/user-manual.md
Support email for questions	ortin@uniovi.es

1. Introduction

"Big code" is a recent line of research that brings together big data and source code analysis [1]. It is based on using the source code of millions of programs to build different types of tools to improve software development [2]. Machine learning is used to create useful predictive models that learn common patterns from

* Corresponding author at: University of Oviedo, Computer Science Department, Federico Garcia Lorca 18, 33007, Oviedo, Spain.
E-mail addresses: ortin@uniovi.es (Francisco Ortin), escaladajavier@uniovi.es (Javier Escalada).
URLs: http://www.reflection.uniovi.es/ortin (Francisco Ortin), https://www.javierescalada.es (Javier Escalada).

https://doi.org/10.1016/j.softx.2021.100711

a large number of source code applications [3,4]. For example, JavaScript programs are used to train a model capable of deobfuscate variable names from their usage [5]; Java and C# files with the same behavior are used to learn automatic translation between these two languages [6]; and vulnerable C source code is used to train models that predict vulnerabilities by analyzing the source code of new programs [7].

Likewise, the Mining Software Repositories (MSR) field analyzes the rich data available in source code repositories to uncover information about software systems and projects [8]. In this case, the source code data is enriched with other information taken from defect tracking systems, archived communications between project personnel, version control systems, and question-and-answer sites [9]. Examples of MSR projects include software repair models that analyze bug fix transactions in software repositories [10], change prediction systems that identify the code prone to change in subsequent releases [11], and the automatic retrieval of help information for source code fragments using question-and-answer websites [12].

One of the languages used to build those machine learning models is the C programming language. From its creation in the 70s, C is still in use, particularly for the development of systems software, embedded system applications, and programs that access specific hardware addresses [13]. Its low demand for runtime system resources and its wide availability have made it a usual candidate to implement language interpreters and computationally intensive programs. According to the Tiobe [14], LangPop [15], and the Transparent Language Popularity Index [16] programming language rankings, C is still the most widely used language in January 2021[1], obtaining the fifth position in the PYPL [17], Redmonk [18] and Trendy Skills [19] rankings.

There exist many different variants of the C programming language, which include language extensions and modifications depending on the operating system, compiler and target hardware. Therefore, different ANSI/ISO standardizations of C are defined to facilitate the development of portable software [20]. However, it is still difficult to find applications written in 100% standard C source code that could be compiled with many different compilers. Most of the existing open-source applications have particular dependencies on non-portable code. This is an issue when building predictive models from source code, since a large number of programs is usually required [21].

There exist tools capable of generating random C source code, but they are mainly aimed at testing compilers, rather than creating machine learning models [22]. Therefore, they are not designed to build massive amounts of standard C code, and they do not cover every language construct—we detail them in Section 2 and evaluate them in Section 5.

For this reason, we developed Cnerator, a Python application that generates large amounts of standard ANSI/ISO C source code [20] to train machine learning models. Cnerator is highly customizable to generate all the syntactic constructs of the C language, necessary to build accurate predictive models with machine learning algorithms. The code it generates is ready to be compiled by any standard language implementation. Cnerator has been used to improve state-of-the-art decompilers [23] and to implement an infrastructure for the automatic extraction of code patterns [24]. The stochastic generation of source code programs has also been used to detect bugs in existing compilers [25]. Another potential use of Cnerator is testing whether a compiler implements the ANSI/ISO standard specification correctly.

[1] These rankings measure the popularity of each programming language, using different criteria. For example, the Tiobe language ranking uses 25 search engines to calculate each language index, depending on the number of searches done by users (https://www.tiobe.com/tiobe-index/programming-languages-definition).

The rest of this article is structured as follows. Section 2 details the related work, and the software functionality and architecture are presented in Section 3. An illustrative example is described in Section 4. Section 5 evaluates Cnerator and compares it with related approaches. Conclusions are presented in Section 6.

2. Problems and background

As mentioned, there are tools for the random generation of C code. Most of them are aimed at finding bugs in C compilers, rather than at training machine learning models.

Csmith is a well-known generator of random C programs [25] created as a fork of randprog [26]. Its main purpose is the detection of bugs in C compilers. Generated programs conform to the C99 standard, and they avoid the undefined behavior constructs specified in C99. To find compiler bugs, each generated program is compiled by different compilers and executed. If a checksum of the global variables upon program termination is different from the rest of executions, the compiler that produced that binary has an error (i.e., randomized differential testing). Csmith implements different safety mechanisms such as pointer analysis, bounded loop constructs, and different dynamic checks. Csmith has been used to detect more than 325 errors in existing compilers, including the verified CompCert C compiler [27].

`ldrgen` is a tool for the random generation of C programs to test compilers and program analysis tools [28]. Existing systems generate large amounts of dead code that the compiler reduces to little relevant binary, because dead code is deleted. For this reason, `ldrgen` implements a liveness analysis algorithm during program generation to avoid producing dead code. It is implemented as a plugin for the Frama-C extensible framework [29]. `ldrgen` has been used to detect missed compiler optimizations [30].

YARPGen is a random test-case generator for C and C++ compilers, created to find and report compiler bugs [31]. YARPGen is created to overcome the saturation point reached by existing compiler testing methods, where very few bugs are found. This is not because compilers are bug-free, but rather because generators contain biases that make them incapable of testing specific parts of compiler implementations. YARPGen generates programs free of undefined behaviors without dynamic safety checks, unlike Csmith. Its approach is to implement different static analyses to generate code that conservatively avoids undefined behaviors. It also implements generation policies that systematically skew probability distributions to cause certain optimizations to take place more often. YARPGen has found more than 220 bugs in GCC, LLVM, and the Intel C++ compiler. Those bugs were not previously found by other compiler testing tools.

The family of Orange random C code generators is focused on generating arithmetic expressions [32]. Instead of differential testing, they track the expected values of each test after execution, checking whether the obtained values are the expected ones. The programs generated by Orange generators are safe, avoiding the undefined behaviors of the C programming language. Orange code generators do not include important language features such as control flow statements, structs, arrays or pointers.

Quest is a code generator tool aimed at finding several compiler bugs related to calling conventions [33]. It generates function declarations randomly, and then generates type-driven test cases that invoke each function. A global variable is generated for each parameter and return value. Assertions are used to check that each value received and returned is the appropriate one. Quest avoids undefined behavior by simply not generating potentially dangerous constructs (e.g., arithmetic expressions). It was used to find 13 bugs in 5 different compilers [33].

3. Software framework

In this section, we first describe the main functionalities of Cnerator. Then, we present its architecture and a brief description of each module.

3.1. Software functionalities

These are the main functionalities provided by Cnerator:

1. ANSI/ISO standard C. All the source code generated by Cnerator follows the ISO/IEC 9899:2018 (C17) standard specification [20]. The code strictly follows the syntax grammar, type system and semantic rules of the standard specification. This makes the generated code to be able to be compiled by any compiler implementing the standard.
2. Probabilistic randomness. C language constructs are randomly generated, following different probability distributions specified by the user. For example, it is possible to describe the probability of each kind of statement and expression construct, the number of statements in a function, and the types of their arguments and return values. To this aim, the user can specify fixed probabilities of each element, or use different probability distributions, such as normal, uniform, and direct and inverse proportional.
3. Highly customizable. Many features of the programs to be generated are customizable. Some examples include the types of each language construct, array dimensions and sizes, struct fields, maximum depth of expression and statement trees, number of function parameters and statements, global and local variables, structures of control flow statements, and type promotions, among others—see the detailed documentation [34].
4. Large amounts of code. Cnerator is designed to allow generating large amounts of C source code (see Section 5). One parameter indicates the number of independent compilation units to be created for the output application, so that each unit could be treated as an independent module. This feature, together with the probabilistic randomness, makes Cnerator an ideal tool to build predictive models, because the input programs used to train such models comprise abundant and varied code patterns.

3.2. Software architecture

Fig. 1 shows a UML package diagram describing the architecture of Cnerator. When executing the tool, three types of optional arguments may be passed: command-line arguments, JSON specification files, and Python post-processing traversals. If no parameter is passed, Cnerator creates a random output program, using the default probability values [34]. The generated program consists of a group of compilation units (a pair of `.h` and `.c` files) that can be compiled independently, even though they commonly depend on other compilation units.

As command-line arguments, the user may pass parameters such as the number of output compilation units, probability values of syntactic constructs, and the output directory and file names, among others (all the parameters are detailed in the user manual [34]). The Parameter Processing module takes all the parameters passed by the user and customizes the behavior of Cnerator accordingly.

Cnerator accepts two types of JSON configuration files as parameters (examples are presented in Section 4). The first one allows specifying the probability values and probability distributions of multiple C syntactic constructs. The Probabilities module stores the default probability distributions of all the syntax constructs, and provides different helper functions to facilitate its specification. As shown in Fig. 3 (explained in Section 4), JSON probability specification files permit the use of those helper functions to modify the default probability distributions.

The second type of JSON input allows the user to control the number and characteristics of all the functions to be generated. For example, we can enforce Cnerator to generate a program with as many functions as built-in types in the language, and make each function return an expression of each built-in type. The Controlled Function Generation module interprets the JSON file to drive the process of program generation. To this aim, it asks the main Program Generation module to generate random functions, and discards those not fulfilling the requirements specified in the JSON file. If no function generation file is provided, Program Generation just produces a random program following the existing probability distributions.

The third type of argument is an ordered collection of Python post-processing specification files. When the user wants the output program to fulfill some requirements not guaranteed by the stochastic generation process, these post-processing files can be used to modify the generated code in order to meet such requirements. By following an introspective implementation of the Visitor design pattern [35], the user can specify the traversal of the program representation produced by Cnerator (an example is presented in Section 4). We use the `singledispatch` Python package [36] to traverse program representations.

The Program Representation module is mainly an in-memory representation of Abstract Syntax Trees (AST) [37]. Cnerator produces ASTs modeling the generated program before generating the output code. The AST data structure implements the Interpreter design pattern [38] to convert a program representation into a set of output compilation units [39].

4. Illustrative example

In this section, we show how Cnerator was used to build classifiers for inferring the return type of C functions from binary code, outperforming the existing decompilers [23]. We used Cnerator to produce abundant C source code in order to train supervised machine learning models. Such models are capable of inferring function return types by just analyzing their compiled binary code. As shown in Fig. 2, state-of-the-art decompilers provide 30% F_1-measure[2] for this classification problem [23], while our machine learning model achieves 79.1%.

Initially, we searched for 100% standard C applications in GitHub, Bitbucket and SourceForge, finding 2329 instances (functions) to be used in our classification problem [23]. We then realized that model accuracy could be increased if more programs were added to the dataset (F_1-measure showed a high coefficient of variation). Therefore, we used Cnerator to generate additional programs and included them in the dataset. Fig. 2 shows how F_1-measure of the classifier grows with an increasing number of instances, obtaining a coefficient of variation below 2% for 20,000 functions. Gradient boosting was the classifier with the best performance (accuracy and F_1-measure) out of the 14 machine learning algorithms tested [23].

Fig. 3 shows an excerpt of two of the JSON files used to customize Cnerator. The one on the left overwrites some default probabilities. The first entry (`function_basic_stmt_prob`) defines the probability of building basic statements (i.e., statements not containing other statements, unlike `for` and `switch`), and the second one states that 10% array definitions should initialize their values. These two examples specify fixed probabilities that

[2] Although different compilers were measured, Fig. 2 only shows IDA (Hex-Rays) because it outperforms the rest of decompilers [23].

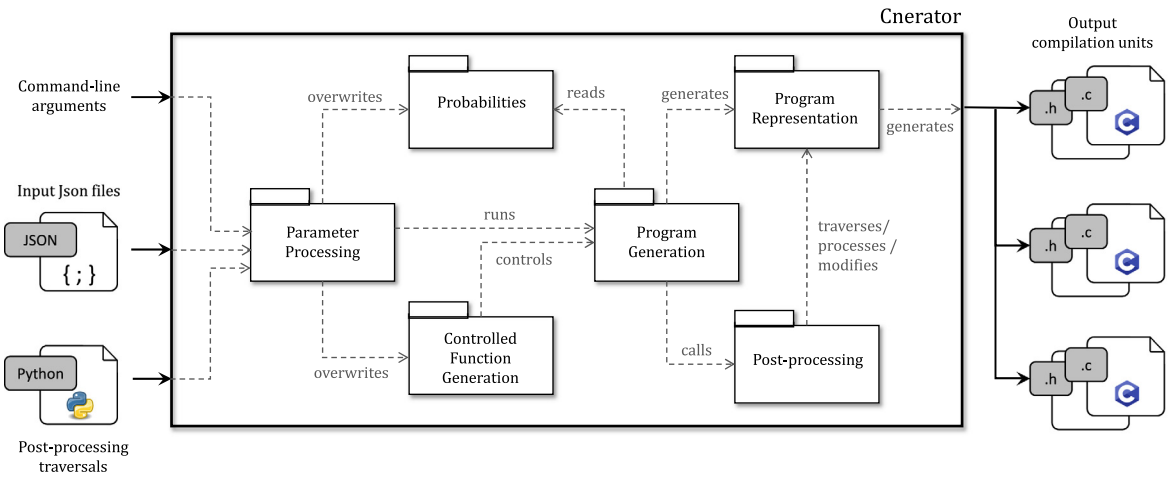

Fig. 1. UML package diagram describing the architecture of Cnerator.

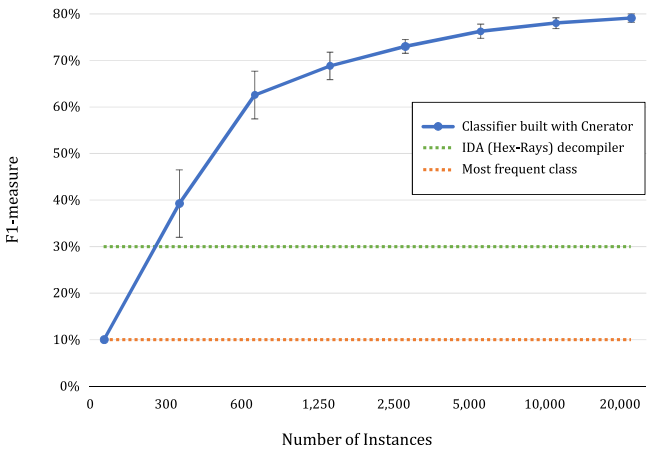

Fig. 2. F_1-measure of gradient boosting classification model for increasing number of instances in the dataset. Whiskers represent 95% confidence intervals, for 30 training/test iterations for each number of instances. The *x*-axis has a logarithmic scale.

must sum zero. The three remaining entries use uniform/equal, proportional and normal distributions to specify, respectively, the usage of primitive types, and the number of function parameters and statements in the main function.

The right-hand JSON file in Fig. 3 shows the controlled function-generation method used to build the dataset for the decompiler scenario. The two first entries are examples of how we made Cnerator generate 1000 functions returning each of the types defined in the standard specification.[3] The condition in the lambda expression checks that the returned type is the expected one. The last entry shows a different example, not used in the decompiler scenario, where the user demands Cnerator to generate a function containing at least one `if` statement with an `else` clause.

Fig. 4 shows an example Python post-process specification file. The code traverses the representation of the generated program (its AST), and adds a unique label before each `return` statement. The purpose of this instrumentation is to identify in the compiled code the binary patterns used for each high-level `return` statement. Those binary code patterns are later labeled with the high-level return type to build predictive models with supervised machine learning algorithms [23].

[3] Only `void` and `bool` are shown for the sake of brevity.

The `_instrument_statements` function takes a list of statements (represented as AST nodes) and adds a unique label – prefixed with `__RETURN__` (line 09) – before each `return` statement. That function is later used in the traversal of function definitions (line 18), and `do`, `while`, `for` and block statements (line 26)—`if` and `switch` control flow statements follow the same template. The code in Fig. 4 is an instance of an introspective implementation of the Visitor design pattern [35]. The `visit` annotations indicate the AST node to be traversed, and default tree traversal is performed with reflection [40].

5. Implementation and empirical results

We compare Cnerator with the random C code generators discussed in Section 2. We select the following evaluation criteria related to the generation of large amounts of standard C code to train machine learning models (Section 3.1). It is worth noting that the comparison is not aimed at identifying the best tool, but at analyzing their appropriateness to generate abundant and varied source code.

1. Generation of standard ANSI/ISO C code, which can be compiled by any standard compiler implementation.
2. It can be specified the probability of each language construct.
3. The tool can be customized to describe properties fulfilled by the different language constructs (e.g., types returned by functions, array dimensions, maximum depth of expression and statement trees, or number of function parameters and statements).
4. Generation of a configurable number of independent compilation units.
5. Avoidance of dynamic undefined and unspecified behaviors.
6. Generation of code for all the language constructs.
7. Generation of large numbers of functions (and their invocations).

The results of the comparison are detailed in Table 1. All the tools but `ldrgen` and Orange generate standard C code. Csmith is the only system, besides Cnerator, that allows setting the probability of some language constructs (inline functions, array accessing loop, and built-in function invocation for Csmith; much more for Cnerator [34]). Csmith supports the customization of some basic properties of the generated code (criterion 3), such as the maximum depth of blocks, pointer indirections, array

```
{
   "function_basic_stmt_prob": {
      "assignment": 0.3,
      "invocation": 0.4,
      "augmented_assignment": 0.15,
      "incdec": 0.1,
      "expression_stmt": 0.05
   },
   "array_literal_initialization_prob": {
      "True": 0.1, "False": 0.9
   },
   "primitive_types_prob": {
      "__prob_distribution__": "equal_prob",
      "__values__": [
         "ast.Bool",
         "ast.SignedChar",
         "ast.UnsignedChar",
         "ast.SignedShortInt",
         …
      ]
   },
   "param_number_prob": {
      "__prob_distribution__":
       ↪                  "proportional_prob",
      "__values__": {
         "0": 1, "1": 2, "2": 3,
         "3": 3, "4": 2, "5": 1
      }
   },
   "number_stmts_main_prob": {
      "__prob_distribution__": "normal_prob",
      "__mean__": 10,
      "__stdev__": 3
   },
   …
}
```

```
{
  "function_returning_void": {
    "total": 1000,
    "condition": "lambda f:
     ↪       isinstance(f.return_type,
     ↪                  ast.Void)"
  },
  "function_returning_bool": {
    "total": 1000,
    "condition": "lambda f:
     ↪       isinstance(f.return_type,
     ↪                  ast.Bool)"
  },

  …

  "function_with_if_else": {
    "total": 1,
    "condition": "lambda f:
     ↪ any(stmt for stmt in f.children
     ↪     if isinstance(stmt, cnerator.ast.If)
     ↪         and any(stmt.else_statements))"
  }
}
```

Fig. 3. Two example JSON files used to customize program generation with Cnerator. The left-hand side shows a sample probability specification file, and the right-hand side specifies an example of controlled function generation.

dimensions and expression complexity. `ldrgen` allows the specification of the maximum number of statements per block, expression depth, functions, function arguments, and statement nesting depth. Cnerator is the only system that allows the generation of any number of independent compilation units.

The feature of avoiding dynamic undefined and unspecified behaviors (criterion 5) shows a different pattern. All the tools but Cnerator provide this feature because their objective, unlike ours, is detecting bugs in compilers (Section 2). On the contrary, Cnerator is the only tool that generates all the language constructs (criterion 6), because the generated code is used to train machine learning models that perform better when the code has more variability. Both Csmith and YARPGen provide a lot of language constructs, but do not support others.[4]

In order to evaluate the capability of generating large amounts of source code (criterion 7), we measure the tools that allow specifying the number of functions in a program (Csmith, `ldrgen` and Cnerator) and ask them to generate programs with an increasing number of functions. Both Csmith and `ldrgen` show a runtime memory error when they are asked to generate 100 functions, in an Intel Core i7 2.5 GHz computer with 16 GB RAM running Ubuntu 20.04.2.0 LTS. The results for Cnerator are detailed in Table 2. Cnerator generates the 20,000 functions for the example in Section 4 in 5.2 min, producing more than 2 million non-empty source lines of code (SLOC).

[4] Csmith does not generate assignments as statements, array-typed struct fields, strings, dynamic memory allocation, floating-point types, unions, recursion, and function pointers. YARPGen does not produce function calls, ++ and −− operators, pointer arithmetic operations, assignments as expressions, non-integer local variables, and has some restrictions when generating floating-point values and loops.

Table 1
Qualitative comparison of random C code generators (● = yes, ○ = no, and ◗ = partially).

Criterion	Csmith	`ldrgen`	YARPGen	Orange	Quest	Cnerator
1	●	○	●	○	●	●
2	◗	○	○	○	○	●
3	◗	◗	○	○	○	●
4	○	○	○	○	○	●
5	●	◗	●	●	●	○
6	◗	○	◗	○	○	●
7	○	○	○	○	○	●

These results show how the differences between Cnerator and the rest of the random C generators are caused by the purpose they are designed for. All the tools but Cnerator are aimed at testing compiler implementations, and that is why, for those tools, the avoidance of dynamic undefined behaviors is so important. However, this feature makes code generation to be more difficult, even requiring the implementation of backtracking algorithms [31]. Moreover, this complexity limits the number of language constructs to be generated, and the production of programs with large amounts of source code. On the contrary, those two last features are very important to Cnerator, designed to train machine learning models. It also provides some other features necessary for its aim, such as the specification of language construct probabilities and a high degree of customization.

```python
from singledispatch import singledispatch
from cnerator import ast

def _instrument_statements(statements: List[ast.ASTNode]) -> List[ast.ASTNode]:
    """Includes a unique label before any return statement"""
    instrumented_stmts = []
    for stmt in statements:                          # iterates through the statements
        if isinstance(stmt, ast.Return):             # if the statement is return…
            label_ast = ast.Label(generate_label())  # creates a new AST node for the label
            instrumented_stmts.append(label_ast)     # and places the label before the return
        visit(stmt)                                  # traverses the statement
        instrumented_stmts.append(stmt)              # appends return after the label
    return instrumented_stmts

@visit.register(ast.Function)
def _(function: ast.Function):
    """Traverses a function definition to add a unique label before each return statement"""
    function.stmts = _instrument_statements(function.stmts)

@visit.register(ast.Do)
@visit.register(ast.While)
@visit.register(ast.For)
@visit.register(ast.Block)
def _(node):
    """Traverses a control flow statement to add a unique label before each return statement"""
    node.statements = _instrument_statements(node.statements)

_return_label_counter = 0

def generate_label() -> str:
    """Generates a new unique label string"""
    global _return_label_counter
    _return_label_counter += 1
    return f"__RETURN{_return_label_counter}__"

…
```

Fig. 4. Python code excerpt of an AST post-processing example.

Table 2
Increasing sizes of C programs generated by Cnerator. SLOC stands for source lines of code and counts non-empty lines of source code, excluding comments.

Number of functions	Seconds	SLOC
10	0.335	1,078
50	0.433	4,159
100	0.696	13,184
500	2.907	73,935
1,000	4.269	139,756
2,500	12.601	342,669
5,000	30.062	717,562
10,000	74.771	1,345,993
20,000	312.237	2,692,157

6. Conclusions

Cnerator is a Python application that provides the controlled stochastic generation of standard ANSI/ISO C code to train machine learning models. It is highly configurable, allowing the user to define the probability distributions of each language construct, specify the properties of the generated functions, perform post-processing modifications of the generated programs, and define the number of output compilation units.

Cnerator has been successfully used to build machine learning models that improve state-of-the-art decompilers [23]. It has also been utilized to implement an infrastructure for the automatic extraction of code patterns [24]. It could also be used to test existing C compilers, including the correct implementation of the ANSI/ISO standard.

Cnerator is distributed with different examples, configuration files and complete documentation. Its source code is available for download at GitHub under a permissive BSD 3-clause license.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has been partially funded by the Spanish Department of Science, Innovation and Universities: project RTI2018-099235-B-I00. The authors have also received funds from the University of Oviedo, Spain through its support to official research groups (GR-2011-0040).

References

[1] Defense Advanced Research Projects Agency. MUSE envisions mining “big code” to improve software reliability and construction. 2014, http://www.darpa.mil/news-events/2014-03-06a.
[2] Allamanis M, Barr ET, Devanbu P, Sutton C. A survey of machine learning for big code and naturalness. ACM Comput Surv 2018;51(4).
[3] Ortin F, Escalada J, Rodriguez-Prieto O. Big code: new opportunities for improving software construction. J Softw 2016;11(11):1083–8.
[4] Ortin F, Rodriguez-Prieto O, Pascual N, Garcia M. Heterogeneous tree structure classification to label java programmers according to their expertise level. Future Gener Comput Syst 2020;105:380–94.
[5] Raychev V, Vechev M, Krause A. Predicting program properties from “big code”. In: Proceedings of the 42nd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages. POPL ’15, New York, NY, USA; 2015, p. 111–24.

[6] Karaivanov S, Raychev V, Vechev M. Phrase-based statistical translation of programming languages. In: Proceedings of the 2014 ACM international symposium on new ideas, new paradigms, and reflections on programming & software. Onward! 2014, New York, NY, USA: ACM; 2014, p. 173–84.
[7] Yamaguchi F, Lottmann M, Rieck K. Generalized vulnerability extrapolation using abstract syntax trees. In: Proceedings of the 28th annual computer security applications conference. ACSAC '12, New York, NY, USA: ACM; 2012, p. 359–68.
[8] Herzig K, Zeller A. Mining your own evidence. In: Making software. O'Reilly Media, Inc.; 2010.
[9] Jung W, Lee E, Wu C. A survey on mining software repositories. IEICE Trans Inf Syst 2012;E95.D(5):1384–406.
[10] Martinez M, Monperrus M. Mining software repair models for reasoning on the search space of automated program fixing. Empir Softw Eng 2015;20(1):176–205.
[11] Malhotra R, Khanna M. Software change prediction: A systematic review and future guidelines. e Inf Softw Eng J 2019;13(1):227–59.
[12] Ponzanelli L, Bavota G, Di Penta M, Oliveto R, Lanza M. Mining stackoverflow to turn the IDE into a self-confident programming prompter. In: Proceedings of the 11th working conference on mining software repositories. MSR 2014, New York, NY, USA: Association for Computing Machinery; 2014, p. 102–11.
[13] Meyerovich LA, Rabkin AS. Empirical analysis of programming language adoption. In: Proceedings of the international conference on object-oriented programming systems languages & applications. OOPSLA '13, New York, NY, USA: Association for Computing Machinery; 2013, p. 1–18.
[14] Tiobe. Tiobe programming language index for january 2021. 2021, https://www.tiobe.com/tiobe-index.
[15] LangPop. Programming language popularity. 2021, http://65.39.133.14.
[16] SourceForge. The transparent language popularity index. 2021, http://lang-index.sourceforge.net.
[17] PYPL. Popularity of programming languages. 2021, https://pypl.github.io.
[18] Redmonk. The redmonk programming language ranking. 2021, https://redmonk.com/sogrady/2020/07/27/language-rankings-6-20.
[19] Trendy Skills. Extracting skills that employers seek in the IT industry. 2021, https://trendyskills.com.
[20] ISO. ISO/IEC 9899:2018 - information technology – programming languages – c. 2018, https://www.iso.org/standard/74528.html.
[21] Babii H, Janes A, Robbes R. Modeling vocabulary for big code machine learning. 2019, http://arxiv.org/abs/1904.01873.
[22] Chen J, Patra J, Pradel M, Xiong Y, Zhang H, Hao D, Zhang L. A survey of compiler testing. ACM Comput Surv 2020;53(1).
[23] Escalada J, Scully T, Ortin F. Improving type information inferred by decompilers with supervised machine learning. 2021, http://arxiv.org/abs/2101.08116.
[24] Escalada J, Ortin F, Scully T. An efficient platform for the automatic extraction of patterns in native code. Sci Program 2017;2017:1–16.
[25] Yang X, Chen Y, Eide E, Regehr J. Finding and understanding bugs in c compilers. In: Proceedings of the 32nd ACM SIGPLAN conference on programming language design and implementation. PLDI '11, New York, NY, USA: Association for Computing Machinery; 2011, p. 283–94.
[26] Eide E, Regehr J. Volatiles are miscompiled, and what to do about it. In: Proceedings of the 8th ACM international conference on embedded software. EMSOFT '08, New York, NY, USA: Association for Computing Machinery; 2008, p. 255–64.
[27] Leroy X. Formal verification of a realistic compiler. Commun ACM 2009;52(7):107–15.
[28] Barany G. Liveness-driven random program generation. In: Fioravanti F, Gallagher JP, editors. Logic-based program synthesis and transformation. Cham: Springer International Publishing; 2018, p. 112–27.
[29] Cuoq P, Kirchner F, Kosmatov N, Prevosto V, Signoles J, Yakobowski B. Frama-c. In: Eleftherakis G, Hinchey M, Holcombe M, editors. Software engineering and formal methods. Berlin, Heidelberg: Springer Berlin Heidelberg; 2012, p. 233–47.
[30] Barany G. Finding missed compiler optimizations by differential testing. In: Proceedings of the 27th international conference on compiler construction. CC 2018, New York, NY, USA: Association for Computing Machinery; 2018, p. 82–92.
[31] Livinskii V, Babokin D, Regehr J. Random testing for c and c++ compilers with YARPGen. In: Proceedings of the ACM on programming languages. Vol. 4, (OOPSLA). New York, NY, USA: Association for Computing Machinery; 2020.
[32] Nagai E, Awazu H, Ishiura N, Takeda N. Random testing of C compilers targeting arithmetic optimization. In: Proceedings of workshop on synthesis and system integration of mixed information technologies, SASIMI '12, 2012, p. 48–53.
[33] Lindig C. Random testing of c calling conventions. In: Proceedings of the sixth international symposium on automated analysis-driven debugging. AADEBUG'05, New York, NY, USA: Association for Computing Machinery; 2005, p. 3–12.
[34] Ortin F, Escalada J. Cnerator user manual. 2021, https://github.com/ComputationalReflection/Cnerator/blob/main/user-manual.md.
[35] Ortin F, Quiroga J, Redondo JM, Garcia M. Attaining multiple dispatch in widespread object-oriented languages. Dyna 2014;81(186):242–50.
[36] Langa L. Singledispatch python package 3.4.0.3. 2021, https://pypi.org/project/singledispatch/.
[37] Rodriguez-Prieto O, Mycroft A, Ortin F. An efficient and scalable platform for java source code analysis using overlaid graph representations. IEEE Access 2020;8:72239–60.
[38] Erich G, Richard H, Ralph J, John V. Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional Computing Series; 1995.
[39] Ortin F, Escalada J. Cnerator developer manual. 2021, https://computationalreflection.github.io/Cnerator.
[40] Ortin F, López B, Pérez-Schofield JBG. Separating adaptable persistence attributes through computational reflection. IEEE Softw 2004;21(6):41–9.