Program Synthesis and Vulnerability Injection Using a Grammar VAE

Leonard Kosta[1, 2*], Laura Seaman[2], Hongwei Xi[1]

[1] Boston University, 111 Cummington Mall, Boston, Massachusetts 02215, USA.
[2] Draper, 555 Technology Square, Boston, Massachusetts 02139, USA.

* Corresponding author. Tel.: 1-617-353-2000; email: kostaleo@bu.edu
Manuscript submitted January 18, 2019; accepted April 6, 2019.
doi: 10.17706/jsw.14.6.227-246

Abstract: The ability to automatically detect and repair vulnerabilities in code before deployment has become the subject of increasing attention. Some approaches to this problem rely on machine learning techniques, however the lack of datasets—code samples labeled as containing a vulnerability or not—presents a barrier to performance. We design and implement a deep neural network based on the recently developed Grammar Variational Autoencoder (VAE) architecture to generate an arbitrary number of unique C functions labeled in the aforementioned manner. We make several improvements on the original Grammar VAE: we guarantee that every vector in the neural network's latent space decodes to a syntactically valid C function; we extend the Grammar VAE into a context-sensitive environment; and we implement a semantic repair algorithm that transforms syntactically valid C functions into fully semantically valid C functions that compile and execute. Users can control the semantic qualities of output functions with our constraint system. Our constraints allow users to modify the return type, change control flow structures, inject vulnerabilities into generated code, and more. We demonstrate the advantages of our model over other program synthesis models targeting similar applications. We also explore alternative applications for our model, including code plagiarism detection and compiler fuzzing, testing, and optimization.

Key words: Abstract syntax tree, grammar variational autoencoder, program synthesis.

1. Introduction

Software analysis has become an important part of the code development pipeline. Developers use software analysis tools to help identify potential vulnerabilities in code so that they can be repaired before launch. Such tools operate either statically on source code or dynamically on instruction traces. The former are generally preferable since they cost less time and do not require that source code be compiled. It is also very difficult to cover the entire code base using dynamic analysis, whereas with static analysis such a feat can be accomplished with much more ease. Therefore, there has been an increase in demand for static analysis tools that can identify vulnerabilities with high accuracy. In order to increase this accuracy, some new static analysis tools have incorporated machine learning techniques into their approaches. This is the goal of the MUSE project at Draper. One of the main difficulties of the project was identifying or creating labeled datasets on which to train the neural networks. Each example in the dataset would need to be a C function labeled as containing a vulnerability or not. Identification of vulnerabilities in the wild is a difficult problem; compiling a corpus of labeled functions from existing code bases at the scale required to train machine learning models is simply not feasible. The ultimate goal of our research is to create a system that allows us to generate an arbitrary number of C functions that we know for certain contain or do not contain

vulnerabilities. The model will support MUSE by enabling the synthesis of large datasets of realistic-looking C functions labeled as good or bad with respect to vulnerabilities. It can also be useful in other applications that require a large number of valid C programs that obey certain constraints. We design and implement a system that meets these requirements and opens the way for future research in this space.

We address 2 questions in this work. First, can we use a Grammar VAE to improve and/or expand upon previous methods of program synthesis in support of the MUSE project? Second, can we enforce constraints on code synthesized by our network? We answer both questions in the affirmative. Many of the previous attempts at program synthesis have either relied on a detailed program specification from the user or a set of input/output pairs in order to synthesize a single function. In order to synthesize a large number of functions, a significant amount of user involvement is required. We circumvent this obstacle by taking a different approach to the program synthesis task: rather than generate one program at a time where each accomplishes a specific task or computes a specific function, we generate many programs that are similar to a set of input functions and that obey certain user-specified constraints. Output functions are not required to solve any particular problem. To generate the output functions, we chose to implement a Grammar VAE. A Grammar VAE is a generative neural network model in which each raw input example is decomposed into a sequence of production rules according to some grammar. During the training procedure, the Grammar VAE encodes each function into the latent space, then decodes each latent vector back into a sequence of production rules corresponding to a valid C function. The input and output sequences are intended to match as closely as possible. After training has been completed, new sequences can be discovered by decoding arbitrary vectors from the latent space. Some logic in the decoder ensures that each output sequence produced by the Grammar VAE is syntactically (but not necessarily semantically) valid. Thus, we can use the Grammar VAE to generate an arbitrary number of syntactically valid C programs that are similar to the functions on which the network was trained. To the best of our knowledge, we are the first researchers to apply a Grammar VAE to the problem of program synthesis. To accomplish our second goal, we add several additional layers of logic to the Grammar VAE in the decode phase. This logic forces the output sequences to behave according to user-specified constraints. The constraints allow control over the kinds of design decisions that programmers would normally make: function return type, arguments, variable types, control flow, and semantic validity. They also allow users to inject vulnerabilities into functions. In summary, we meet the goals of using a Grammar VAE for program synthesis and enforcing constraints on the synthesized code. We construct an appropriate grammar and handle issues of context-sensitivity during sequence generation; we generate a dataset on which to train our model; we apply the Grammar VAE to the problem of program synthesis; we train and tune the model extensively; we analyze the latent space; and we impose constraints on generated functions.

2. Background

2.1. Program Synthesis

Automatic program synthesis has long been the subject of research. The goal of program synthesis is to generate full or partial programs from a specification [1]. Specifications vary based on application. Often, they involve a set of rules or statements in a formal logic, but some recent works use input/output pairs as formal specifications [2]. Early attempts at code synthesis leveraged theorem provers to construct a proof of the specifications, and then build a program to satisfy the proof [3]-[5]. The next popular approach was transformation-based synthesis, which iteratively transformed the formal specification into the desired output program [6]. One shortcoming of these deductive approaches is that it is left to the user to provide the specifications that precisely define the behavior of the desired function. Specifications of this kind can be even more difficult to compose than the desired code would be [1]. Users must have a deep understanding of

the underlying logic of the program to be generated, and this is not always achievable.

Deductive synthesis gave way to inductive synthesis based on input/output examples, program sketches, and other forms of incomplete specification. The advantage of this methodology is that it does not require users to have much or any understanding of the underlying logic of the desired program. Machine learning has been applied to the inductive synthesis problem with impressive results. Inductive synthesis does not usually work directly with source code. The synthesized programs are often latent (hidden in the weights of a neural network or the DNA of a genetic algorithm, for instance) or written in a domain-specific language in which syntax is not a concern. A common example of the latter case is when the synthesizer must determine how to string together calls from a programmer-defined pool of possible commands, which can be executed successfully in arbitrary order. Some machine learning techniques fall outside of what is traditionally understood as inductive synthesis. Cummins *et al.* [7], for instance, attempt to write code one character at a time using a neural network. One limitation with this and similar models is that, in general, there is no guarantee that the generated code will be either syntactically or semantically correct.

We take program synthesis in a slightly different direction than those explored by most of the previous work. Synthesized programs are usually very specific—they are intended to produce a certain output, solve a particular problem, or model some function. We attempt to generate many programs that do not necessarily solve any particular problem. This methodology is better suited to the applications that we target.

2.2. Grammars

Grammars are formal constructs that describe the set of all valid strings in a language. In this work, we consider two types of grammars: context-free and context-sensitive. A Context-Free Grammar (CFG) G is formally defined as a 4-tuple $G = (V, \Sigma, R, S)$ where V is the set of non-terminal symbols, Σ is the set (disjoint from V) of terminal symbols, R is the set of production rules (also called the vocabulary), and S is the start symbol. The rules in R describe how to transition from non-terminal symbols to strings in the language. When written in Backus-Naur Form (BNF), each rule is of the form $A \rightarrow \alpha$ where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$. The string is complete when it has been fully expanded so that it contains only terminal symbols.

A Context-Sensitive Grammar (CSG) is defined by the same 4-tuple as a CFG, but with the additional requirement that every rule be of the form $\alpha A \beta \rightarrow \gamma$ where $A \in V$; $\alpha, \beta, \gamma \in (V \cup \Sigma)^*$; and $|\alpha A \beta| \leq |\gamma|$. A CSG allows for each application of a rule to take into account the context of terminal and non-terminal symbols in the string, so CSGs can describe more complicated languages than CFGs can. Most modern programming languages are not context-free, which is why we must extend the original Grammar VAE architecture. Throughout this work, we say that a C program is "syntactically valid" if it parses into an abstract syntax tree without errors. We say that a program is "semantically valid" if it both parses and compiles without errors.

Kusner *et al.* [8] introduce the Grammar VAE, an extension to the traditional VAE model that integrates a grammar during the encoding and decoding phases. When the input sequences can be written in terms of a grammar, the Grammar VAE learns a coherent latent space and makes the guarantee that all sequences generated by the VAE will be syntactically valid according to the grammar. The original Grammar VAE supports only CFGs; we extend this model to CSGs by incorporating context information. Kusner *et al.* [8] used a Grammar VAE to learn a latent space for molecules according to the SMILES molecular CFG [9]. The benefits of the Grammar VAE over other generative models were first, that every molecule extracted from the latent space was syntactically valid; and second, that the latent space was more meaningful, allowing for much quicker search for desirable molecules. After training, the Grammar VAE was used to generate new molecules by sampling latent vectors from a multivariate Gaussian distribution and decoding them. Similarly, we use our Grammar VAE to generate an arbitrary number of semantically valid C functions.

3. Methods

3.1. System Overview

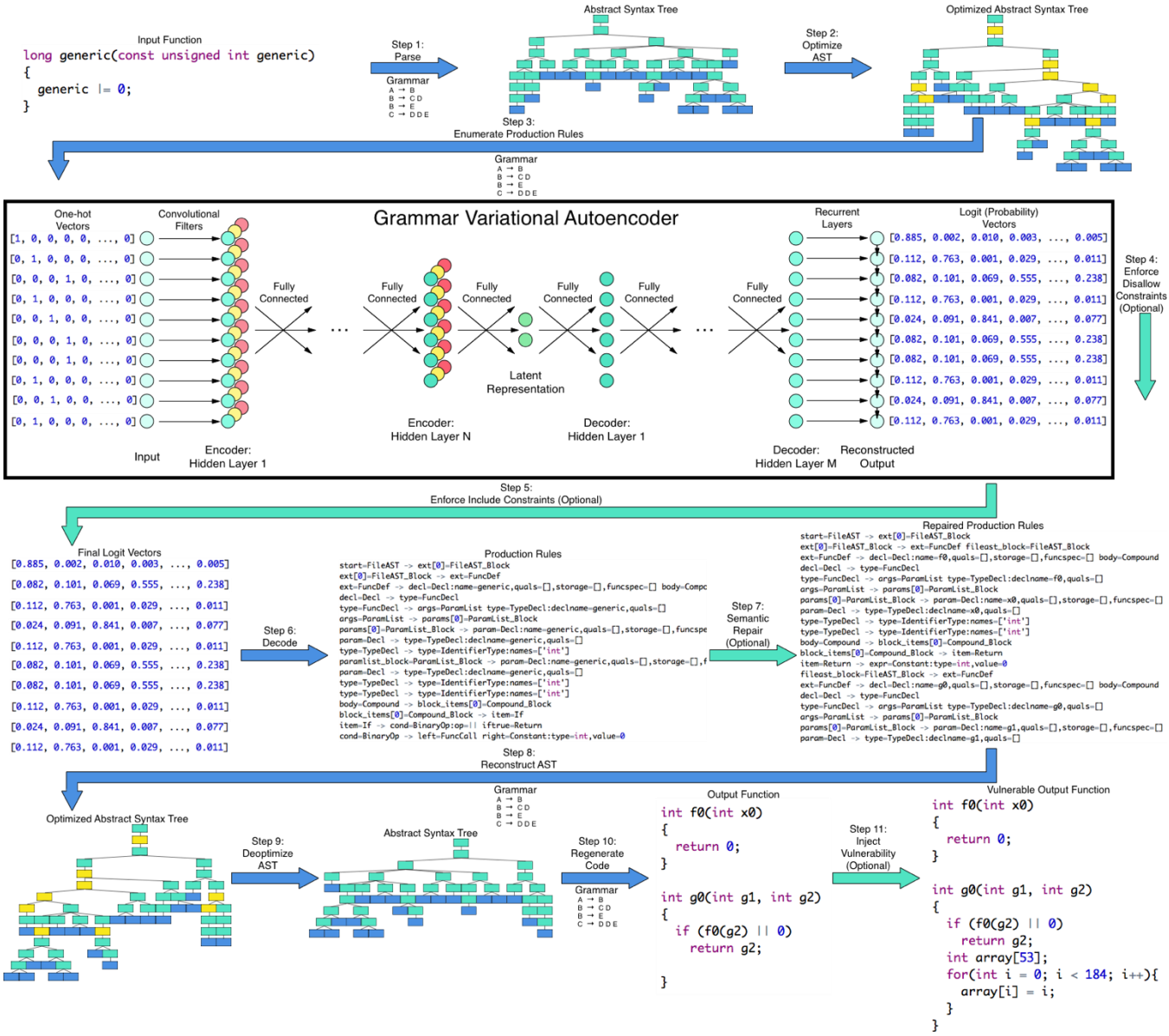


Fig. 1. System overview.

Fig. 1 gives an overview of our system. There are two primary points of entry into the system. The first is with a C function in the training dataset as shown in the top left of the Fig. The system takes the function and parses it into an abstract syntax tree. We wish to reduce the number of unique production rules required to represent all of the functions in our dataset because a smaller vocabulary produces smaller one-hot vectors, and smaller one-hot vectors improves the neural network's performance. Thus, to reduce the number of unique production rules required to represent any set of input functions, the system optimizes the abstract syntax tree by making it near-binary. In the next step, the system outputs the sequence of production rules that correspond with a preorder traversal of the abstract syntax tree and converts each rule into a one-hot vector using the grammar. The boxed subsystem represents the Grammar VAE. The neural network attempts to reproduce the input from the limited information in the latent vector with as high a degree of accuracy as possible. The sequence of one-hot vectors representing the input function is passed to the neural network.

The neural network encodes the sequence through several layers. The first layers in the network are convolutional layers, which learn position-invariant relationships within the input. After the convolutions are applied, the network passes the input through several fully connected layers until it becomes a compressed latent vector. The latent vector is the second point of entry for the system. In order to generate new functions not seen in the training dataset, one can choose a random latent vector and send it through the neural network decoder and the remainder of the system. This point of entry makes the system useful as a generative, program synthesis model. Regardless of the point of entry chosen, the latent vector is decoded through several layers in an attempt to reproduce the input. The last of the decode layers are recurrent layers, which specialize in learning sequential relationships in data. The output of the final recurrent layer is a sequence of logit vectors, each of which gives the neural network's prediction for the production rule at that index in the sequence. The system then enforces the first two sets of optional user-provided constraints. The first class of constraints enforced is the disallow constraints, which are applied as the sequence of logit vectors is first decoded. The second class is the include constraints, which are applied both during and after the decoding of the initial sequence. The result of these transformations is the sequence of final logit vectors that correspond to the neural network's best guess of the original input sequence given the optional user-provided constraints. The system then completes the decode phase by translating the logit sequence into production rules in the grammar. The Grammar VAE architecture ensures that the decoded production rule sequence corresponds to a syntactically valid function according to the grammar. Depending on user-provided repair constraints, the last class of constraints, the production rule sequence may undergo semantic repair, which transforms the syntactically valid function into a fully semantically valid function. The system will then use the grammar to reconstruct the optimized abstract syntax tree from the production rules. From the optimized abstract syntax tree, the system derives the corresponding deoptimized abstract syntax tree, which is turned into C source code using a C parser. If the user has specified that the function should contain a vulnerability, one is injected at this point. Vulnerability injection is another optional repair constraint. The system now returns the function produced as output. Table 1 lists the subsystems used in each step of Fig. 1.

Table 1. Subsystems Required for Each Step

Step	Subsystems required
1	Any C parser
2	CSG construction, AST optimization
3	CSG construction
4	Grammar VAE, disallow constraints
5	Include constraints
6	Grammar VAE
7	Repair constraints
8	AST reconstruction
9	AST optimization
10	Any C parser
11	Repair constraints

3.2. CSG Construction

One key requirement for our model is a grammar. There are several available grammars for C, most notably that found in the appendix of Kernighan *et al.* [10], however many of the context requirements are conveyed through explanation in plain English and not BNF. In fact, many grammars treat C as a context-free language, an assumption that can be proven false by the well-known typedef-name context requirement at the least. Thus, part of our work involved composing a suitable CSG in BNF for at least a subset of the C language. The grammar must be in BNF so that each rule can be represented as a one-hot vector.

It was beyond the scope of our work to write such a grammar by hand. However, because the sequence of production rules used to generate a piece of code can be extracted directly from a preorder traversal of the associated AST, we can reverse engineer the grammar from a large enough pool of C functions. During enumeration, we can include context information in each rule by listing the successor and predecessor nodes in the AST, but we do not know how many symbols in general we must include in each rule to have the correct level of context required to ensure the syntactic validity of output functions. Two further considerations: only past context information can be incorporated into the model, since the Grammar VAE only knows predecessor symbols when it outputs a given rule; and as the level of context increases, the vocabulary size increases exponentially. Thus, we decided to use one step of past context information (i.e. one predecessor node in the AST) in each rule. This kept the vocabulary sufficiently small to allow the training process to succeed, but sufficiently expressive to guarantee the syntactic validity of output sequences. We also limited the size of the vocabulary by performing several transformations on the functions; most importantly, we replaced all identifier names and literals with generic values.

We built our vocabulary in this manner using the production rules found in 100 randomly selected functions from the MUSE C source dataset. Our final vocabulary contained 234 rules, one of which was for padding.

3.3. AST Optimization

We would like to modify function ASTs in such a way that we reduce the number of production rules in the grammar and preserve AST correctness. We will accomplish this by making the AST near-binary. Specifically, we transform any node in the AST that accepts a variable number of children (e.g. a block statement) into a linked list of nodes that each accept at most two children: a value node, and a next node. This has 3 important effects. First, it increases the number of nodes in each AST by a small factor; this is not an issue for our applications. Second, it dramatically reduces the number of production rules in the vocabulary when each rule is discovered through the enumeration process mentioned above. Third, it allows the Grammar VAE to reach a wider range of output functions. Without optimization, the Grammar VAE would not be able to output a function with a block containing, say, 4 statements unless it had seen a function with a block of exactly that length in the training dataset. Optimization is an invertible procedure, so deoptimization is trivial given the optimization algorithm.

3.4. Grammar VAE

We use the work of Kusner *et al.* [8] as a starting point for the design and implementation of our own Grammar VAE. We refer to their Grammar VAE explicitly as the Molecule Grammar VAE and ours as just the Grammar VAE. We implement two main models: the Grammar VAE and the Character VAE. The Grammar VAE takes as input a sequence of production rules that comprise the input function and returns as output a sequence of production rules that correspond with a syntactically valid C function. Our model makes hard guarantees about the syntactic validity of outputs, and has the ability to enforce user-provided constraints on generated functions. The Molecule Grammar VAE uses a grammar with 76 production rules and a maximum sequence length of 277. The Grammar VAE's grammar has 234 production rules, and we use a maximum sequence length of 50. The VAE is split into two components: the encoder, comprising all layers up to and including the latent vector; and the decoder, beginning with the latent vector and containing all remaining layers. The purpose of the encoder is to compress the input data into the latent vector in such a way that the important information is retained and the input can still be reconstructed. The encoder begins with several 1-dimensional convolutional layers. These convolutions help our neural network learn position invariant relationships between particular sets of rules. Information from the filters is passed to the subsequent dense layers. The latent vector is the most compact layer, containing just 10 neurons. Whereas the encoder

compressed the input, the decoder aims to recreate the input sequence using only the information available in the latent vector. The first layers in the decoder are expanding dense layers. Then come several recurrent layers. These layers take advantage of sequential patterns in the data to improve reconstruction accuracy.

The Character VAE is a simple VAE that attempts to write code one character at a time. The Character VAE takes as input a sequence of characters that comprise the input function and returns as output a sequence of characters that should correspond with the input C function. The characters are one-hot encoded. The internal architecture of the Character VAE is similar to that of the Grammar VAE: convolutional layers, dense layers, latent vector, dense layers, and recurrent layers. Consistent with previous work, we use this model as a baseline for our experiments.

When Kusner *et al.* [8] introduced the Grammar VAE, they suggested that the architecture could be applied to the problem of program synthesis. However, as introduced, their architecture was not readily applicable to program synthesis for two main reasons. First, although the Molecule Grammar VAE produced only syntactically valid outputs, it had a high rejection rate of about 70%. When generating a large number of outputs, the rejection rate must be much lower. This discrepancy is not thoroughly explained, but we speculate that it is mainly due to the fact that decoded sequences that meet the maximum sequence length but do not yet represent a complete molecule are automatically rejected. We solve this problem by introducing a sequence completion procedure. If the maximum sequence length is reached, but the function is not complete, the decoder passes control to the sequence completion procedure, which uses the grammar to find the remaining production rules. By default, the rule with the fewest children that does not violate syntactic validity is chosen, although randomness can be incorporated to increase output function variety. The second reason why the original Grammar VAE was not ready for program synthesis is that it was only designed to handle CFGs. As stated earlier, most modern programming languages are context-sensitive, and attempts to treat the grammar as context-free in a Grammar VAE setting result in a very high number of syntactically invalid outputs. This is a problem if one would like to generate a large corpus of functions, as we do for applications like MUSE. We address this problem by incorporating context into the rules themselves, making context-sensitivity checks during the decode phase, and reconstructing the AST using the context information available in the production rules. As a result of our changes, 100% of output sequences correspond with syntactically valid functions, which is important for generating large datasets.

As in a context-free environment, the grammar is incorporated into the VAE decoder. Each logit vector output by the neural network is translated into a production rule sequentially. First, an empty stack of non-terminal symbols is instantiated. The start rule in the grammar is automatically chosen as the first rule in the sequence, and the rule's child non-terminal symbols are pushed onto the stack. For each next logit vector in the sequence, only those transitions in the grammar that begin with the non-terminal symbol on top of the stack are allowed. This is enough to guarantee the syntactic validity of outputs when using a CFG. Given the probabilities in the logit vector, the next rule is chosen, the symbol on top of the stack is popped, and any child non-terminal symbols of the new rule are pushed. This process continues until there are no more symbols on the stack (a valid output) or the maximum sequence length is reached. Our sequence completion procedure continues with the same logic.

With the introduction of a CSG, extra logic is necessary when each next rule in the sequence is chosen. Given our relatively small vocabulary size, there were relatively few (a few dozen) syntactically invalid structures that appeared in various functions. Thus, it was possible to search for these cases and add specific logic to prevent each. Table 2 gives several examples of invalid structures caused by failure to observe context-sensitivity. Case-by-case logic is not necessary if the full grammar can be written in BNF, so including a greater degree of context in rules could solve this automatically. However, as stated earlier, we do not know how many steps of context would be required to do this, and each additional step causes an exponential increase in

vocabulary size.

Table 2. Selected Context-Sensitivity Issues

Invalid structure	Valid structure	Context-sensitivity issue
int f() = 0{}	int f(){}	Declaration node cannot have “init” if child of Function Declaration node
(char *)x;	char *x;	Pointer Declaration node cannot have “declname” if child of Cast node
int f(x){}	int f(int x){}	Parameter List node cannot use identifiers if child of Declaration node
int f(){int g(){}}	int f(){}	Function Declaration node cannot be child of Function Declaration node
case:	case: f();	Final Case node in switch statement cannot be a leaf node

The MUSE dataset consists of over 11,000,000 C functions. Every production rule used in every function of our training dataset must be in our vocabulary. Because we must include context in each production rule, the number of rules found in the full MUSE dataset is well over 30,000. A vocabulary of this size is good for expressiveness, but has two main disadvantages: physical size and neural network performance. Using such large one-hot vectors, the size of the training dataset was well over 1TB. Not only was it a burden to store such a large file, but training took on the order of weeks for each model. Additionally, the neural networks never learned a good latent representation of the data, and were therefore useless as generative models.

We needed to limit the vocabulary size somehow. We accomplished this by randomly selecting 100 functions from the MUSE dataset and extracting their production rules; we took these and trimmed them to 234 based on which we felt would be used most often by programmers. With these rules, we randomly generated 250,000 functions subject to certain constraints. The functions in this dataset used only the production rules that we selected. This gave us a training dataset of sufficient size and with a small vocabulary. We also note that, even across the entire MUSE dataset, there were too few functions that used only the selected production rules to create a suitable training dataset for a deep neural network. Future work should attempt to expand the vocabulary and train on human-written functions. We made some progress toward this goal by relaxing the requirement that all output functions be syntactically valid, but the resulting model did not produce functions of as high quality as those of the model that we present in this work. One major advantage of the Grammar VAE architecture is that it learns the semantic qualities of the training dataset exceptionally well. If we trained the Grammar VAE on the original MUSE dataset, then we could expect the generated functions to share many semantic qualities with those functions. A model capable of such a feat would be an important breakthrough for program synthesis.

We trained our neural network to minimize the ELBO loss function. We experimented with various network architectures and performed hyperparameter tuning. We split our dataset of 250,000 functions into a training dataset of 225,000 functions and a validation dataset of 25,000 functions.

We considered 3 major performance metrics when evaluating the model: training and validation loss and accuracy values, code similarity metrics, and coherence of the latent space. Table 3 shows our model's performance on the training and validation datasets after tuning. Loss and accuracy numbers do not have much intuitive meaning on their own, but they are useful for comparing results between models, and they are the primary method by which we judge the results of training. In addition to loss and accuracy, we consider several code similarity metrics for evaluating, comparing, and validating our models. The purpose behind these supplementary metrics is to provide greater clarity on the success or failure of our trained model by giving an intuitive, but quantitative measure of its performance. Our supplementary metrics are gathered on 1,000 randomly selected functions from the training dataset. Importantly, in this test and all of our experiments, every decoded function parses correctly; out network produces only syntactically valid

functions. This is a major improvement on previous work. In this experiment, we set the variance used in the decoder to 0 so that we can determine how well the model can produce unique functions. Early models produced very few (< 50) unique functions, and we found that the quality of their generated outputs was poor. Increasing the decoder variance by even a small amount will lead to 100% unique functions. We produce about 70% unique functions on average. We also examine how many decoded functions have the same return type and number of arguments as the original input functions. These two metrics are useful for measuring basic semantic similarity. Finally, we calculate the mean and median Levenshtein distance. Levenshtein distance is defined as the number of single character edits required to transform one string into another, and is used in many code plagiarism checkers to determine how closely related two programs are based only on text. Our third performance metric was the coherence of the latent space. Fig. 2 shows an idealized latent space in which like features cluster together and each axis controls a set of semantic properties. In practice, the latent space is not so tidy. We traverse the latent space in randomly selected regions to see, in general, how smooth the transitions are. A good model will have smooth transitions throughout the latent space and similar features will be clustered together.

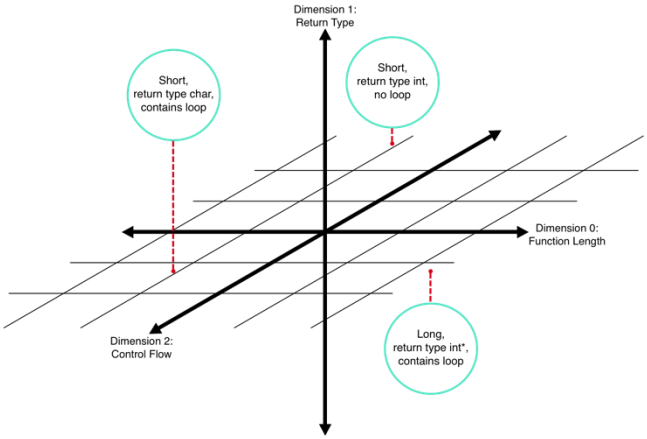


Fig. 2. Idealized latent space.

Table 3. Grammar VAE Training Results

Metric	Final Model Performance
Training loss	0.0461
Training accuracy	0.0338
Validation loss	0.0459
Validation accuracy	0.0339

3.5. Constraints

We would like to guarantee that our generated functions obey certain user-specified constraints so that the user can make some of the design decisions that programmers would normally make about functions: return type, number and types of arguments, control flow, variable types, and more. We also allow the user to specify that the decoded function be semantically valid and/or contain a vulnerability. We divide our constraints into 3 categories based on when they are enforced in our model. The constraint subsystem is another of our major contributions. Without constraints, the user has little control over the kinds of functions that are generated. The user can sample intelligently from the latent space if they know that a certain semantic quality is common in a region of the space, but there are many problems with this approach: search for particular features in the latent space will be at least $O(n^z)$ where n is the number of samples taken along each dimension and z is the dimensionality of the latent vector (here, 10); there is no guarantee that an output with all of the desired

properties even exists in the latent space; and even if such a region exists, it is probable that multiple samples must be taken because the bounds of the region are not well defined. As the number and complexity of semantic qualities desired increases, the search for satisfactory outputs becomes more difficult. These reasons also explain why it is not sufficient to set the variance parameter of the decoder to some non-zero number and repeatedly decode the latent vector until the decoder produces a function with the desired properties. Our constraint subsystem alleviates these concerns and facilitates the search for outputs with specific semantic qualities. With near-perfect success rate, the output will satisfy all given constraints. Users can take advantage of constraints to generate datasets with particular semantic qualities, which can be useful to train or optimize models for different properties (e.g. a compiler that exhibits excellent performance on programs with loops). They can also generate very specific new kinds of outputs. In the case of the Molecule Grammar VAE, one could use constraints to produce new molecules that contain, say, a benzene ring.

The disallow category consists of constraints that are enforced by preventing the decoder from choosing some subset of production rules in certain contexts. All of this logic is contained in the decoder and is executed as each new rule in the sequence is chosen. The decoder marks rules that would otherwise have been valid in a given context as invalid, restricting the choices available to the decoder. Some of the implemented disallow constraints include: return type modifier, argument number modifier, argument type modifier, and control flow modifiers (e.g. no loops). Disallow constraints are by far the simplest to enforce, although doing so is not always trivial. Sometimes certain rules can be marked invalid without context information, but other times the decoder must use previous rules in the decoded sequence to determine whether or not a rule should be marked invalid based on the constraints.

The basic premise of include constraint enforcement is that we prevent the decoder from returning the completed function if it does not contain the desired feature. This process occurs in two phases. First, we mask out any rules that result in the function ending unless we see the desired feature appear in the decoded function. This allows the function to produce the feature organically. For simple features, this frequently occurs. The second phase begins when the maximum sequence length is reached and the function completion procedure begins. Now, we pursue the more aggressive strategy of forcing the function to choose production rules that will lead to the inclusion of the feature by increasing their priority in the queue of rules. Then, we allow the function to end. We could have decided instead to add the feature at the beginning of the function, but our two-step approach increases the variety of decoded functions under include constraints. The constraints that fall in this category influence various control flow features, such as the inclusion of loops, branches, or even specific statements.

The final type of constraint requires us to take a sequence of rules corresponding to an already valid AST and fix them in some way. We need repair constraints in order to make some of the semantic guarantees that cannot be expressed as disallow or include constraints. In particular, we would like to use repair constraints to make guarantees about semantic validity and the presence of vulnerabilities. The advantage of our approach is that, at minimum, every decoded sequence of production rules corresponds to both a complete AST and a syntactically valid function. So, when we do repair, we are not starting from scratch; rather than generate a semantically valid function, we can take a syntactically valid function and make it semantically valid. We find this approach easier because the level of context information required to produce syntactically valid C functions is far less than that required to produce semantically valid ones. We can also use a variety of techniques to solve this problem because for each function we have its production rules, AST, and C source representation. Sometimes we find it convenient to work at one level, sometimes another. However, constraints that fall in this category are very complex, and the implementation of each is a small feat in itself. We implemented two repair constraints: semantic repair and vulnerability injection.

The semantic repair constraint takes a sequence of production rules corresponding to a syntactically valid

function and returns another sequence of production rules corresponding to a semantically valid function. We strive as much as possible to preserve the meaning of the original sequence. The logic for the semantic fixer is quite complicated. Recall that we transformed production rules to use generic identifier names and literal values. In order to guarantee semantic validity, we must replace generic identifier names in such a way that the program obeys variable declaration rules, scoping rules, type rules, etc. The logic of the semantic fixer is similar to that of a typechecker or parser. The semantic fixer uses a namespace, a mapping between variable names and their types, to force variables to obey scoping and type rules. Like a typechecker, the semantic fixer uses recursion to maintain this namespace as variables go in and out of scope. Every declaration adds a numbered identifier to the namespace, and the type is set based on the available context information; when variables are referenced, the most recently used variable in the namespace that is of the correct type is chosen. We use the principle of temporal locality to guide this decision. There are many other changes that must be made to the output function during this process, including: ensuring that all control flow structures appear in valid contexts, replacing invalid unary operations, changing assignments to declarations when necessary, inserting declarations before variable use, adding basic function implementations when other functions are called, and more. After all of these transformations are complete, the result is a semantically valid C function. Our preliminary tests indicated a 100% success rate, but the results of our final experiments show that the semantic repair constraint fails a small percent of the time (< 0.1%).

Vulnerability injection is done after the function has been semantically repaired. In order to inject a vulnerability, the user must define a vulnerability template: a string with special keywords that indicates the structure of a code segment containing the said vulnerability. The keywords describe how special regions of the template are filled. For example, the keyword _CODE_ means that this section can be filled with arbitrary code; the keyword _CONSTANT_ means that the keyword should be replaced with a constant; the keyword _INT_EXPR_ means that the keyword should be replaced by an expression that returns int. In addition to the template, one must define any special rules for filling in the keyword values. For example, in our template for the buffer overflow vulnerability, the buffer size must be strictly less than the loop variable's maximum value in order to trigger the desired error, and both must be strictly greater than 0. Vulnerability injection then proceeds as follows. First, we allow the injector to choose any value for constants as long as they fall within the bounds of some predefined maximum and minimum values. Next, we fill in code blocks with arbitrary pieces of code. We then replace integer expression keywords with random integer expressions. The function that performs this action takes as an argument the variable that is to be used in the expression (e.g. the loop counter). The final step is to actually perform the injection of the vulnerability. We deconflict the namespaces of the function and the vulnerability itself so that the two share no variable references. Then, we choose a random line in the function. We check to see if this would be a satisfactory index at which to insert the vulnerable code. If the vulnerability would interfere with any control flow statements or would compromise the semantic validity of the function, then we sample another index until we find one that will work. Finally, we insert the vulnerability string into the function string at the chosen index and return the result. The output of the vulnerability injection process is a semantically valid, vulnerable function. This process is successful 100% of the time. In future work, we would like to implement more complex vulnerabilities. However, the main reason that we chose to implement vulnerabilities in the manner that we did is because the vulnerable functions that we produce are very similar to those found in the MUSE dataset (which includes a few handwritten functions with labeled vulnerabilities). Also, our approach allows us to train the MUSE classifiers to recognize position invariance both within and between vulnerabilities. Previous work in vulnerability injection worked on the binary level [11]. This process was more sophisticated in some ways, but also had a high failure rate and was far slower, sometimes taking on the order of minutes to inject a single vulnerability.

This delay is not acceptable if one wishes to generate datasets of hundreds of thousands or millions of functions. Our approach works every time, and does not incur any noticeable penalty in terms of function decode time.

3.6. AST Reconstruction

Suppose that we have a sequence of production rules that correspond to the preorder traversal of the AST of a syntactically valid C function (the output of the Grammar VAE) and we would like to regenerate said function. Because the production rules correspond directly to the preorder traversal of the AST, we can reconstruct the AST by making recursive calls on the rule sequence: for every child found in the production rule, we make a recursive call to reconstruct that subtree in the AST. After a child node is reconstructed successfully, we attach it to the current node. The current node is fully reconstructed when all of its child nodes are reconstructed. When the recursion is complete and the root node is reconstructed, we have a complete optimized AST. AST reconstruction will fail if the Grammar VAE naively treats production rules in a context-free manner. During the node reconstruction process, several context-sensitivity checks are made in order to ensure that the resulting AST is that of a syntactically valid function. Based on context information, we know when a production rule belongs to a parent node or child node, which directly corresponds to which of the two nodes will make the recursive call on the rule in question.

Table 4. Constraint Enforcement Statistics

Constraints	Average decode time (s)	Success rate
None	0.246	-
1 disallow	0.272	100%
3 disallow	0.251	100%
1 include	0.834	100%
Semantic repair	0.295	99.93%
Vulnerability injection	0.284	100%

4. Experiments and Results

4.1. Example Generation

We want to determine the extent to which our model can output high quality functions. A high quality function is one that is syntactically and semantically valid, and contains sufficient semantic information to be used as an example in a dataset to train the MUSE classifiers. Our neural network guarantees syntactic validity of outputs, so already we have met one of the three qualifications. Semantic validity is handled by the semantic repair constraint. The final qualification, that the contents of the function must carry some meaning, is difficult both to measure and attain. We would like our generated functions to be indistinguishable from human-written functions, but this is a lofty goal for the scope of our work. Another way that we can measure the success of our model is to train the MUSE classifiers on our generated datasets and see whether performance improves, but this goal is also beyond the scope of our work. As is the case with many generative models, the only way of which we are currently aware to measure program quality is visual inspection. We also have some quantitative metrics to assist in our analysis. Example generation is the most important of our experiments. Our model was designed to create datasets of functions to train the MUSE classifiers, so we need to generate functions of the highest quality possible. We also note that, even if the functions are not yet sophisticated enough for the MUSE project, there are still several applications for which our model would be appropriate. We explore these in Section 6.

To conduct this experiment, we sample and decode 10,000 latent vectors from the multivariate Gaussian distribution $Q \sim N(0, c * I)$ where c is a constant that determines how different the latent vectors are from

each other. This value is a hyperparameter; we tune it over several iterations of the experiment until we produce the best functions possible. Lower values reduce the number of unique functions produced, whereas higher values lead to functions of lower quality.

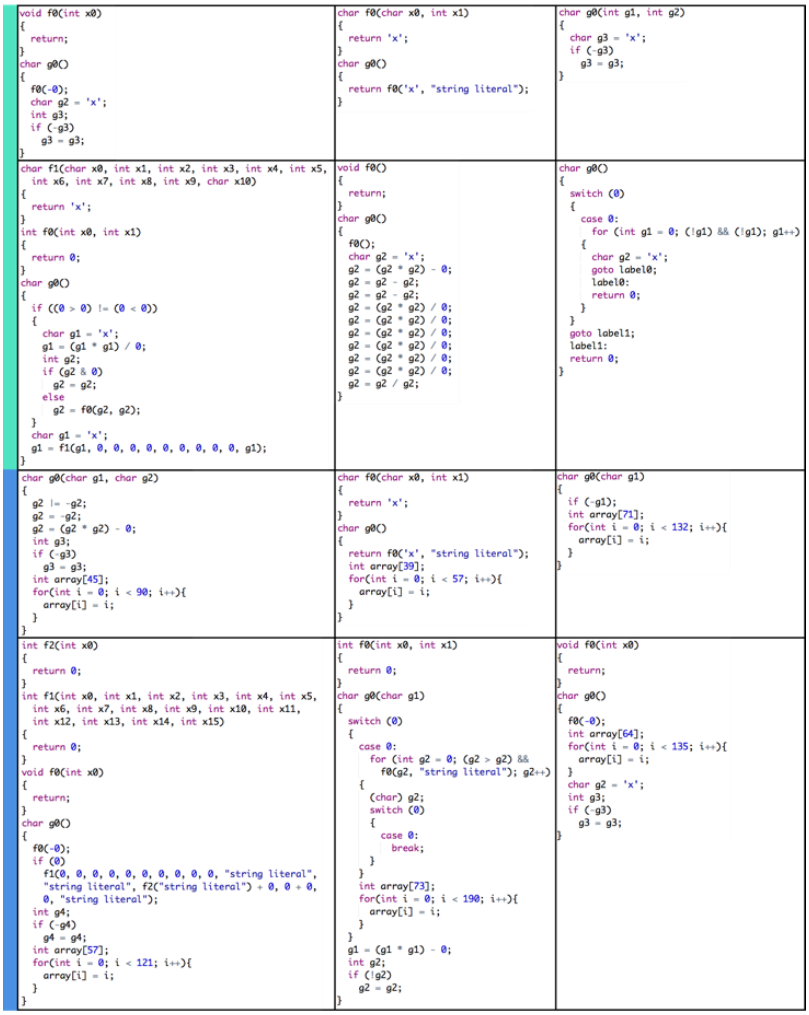


Fig. 3. Grammar VAE generated functions.

We ran the experiment several times using various combinations of constraints and evaluated average function decode time, constraint success rate, and function quality. Table 4 shows the results for these experiments. Average decode time remained fairly consistent, but spiked when we enforced include constraints. The reason for this is that many functions that would have ended quickly without the constraint are not allowed to end until the maximum sequence length is reached, at which point the feature is added to the function. The success rate is perfect for all but the most complicated constraints. Function quality is generally good throughout the experiments, leading us to conclude that our generated functions will be a valuable addition to the MUSE dataset and could be useful in other applications, discussed below. The vulnerabilities in particular look very similar to those found in the MUSE dataset. However, include constraints cause function quality to deteriorate in many functions; the logit sequence may only be meaningful until the index where the function was supposed to end. The remainder of the logit vectors may not contain a high degree of semantic information. Fig. 3 shows the output of the final two experiments: semantic repair (teal) and semantic repair/vulnerability injection (blue). In the final experiment, we used a buffer overflow vulnerability template. The generated functions are not without their stylistic flaws, but

almost all of them are semantically valid functions that compile and execute, users can use constraints to control various semantic features, and the functions contain real vulnerabilities. Moreover, the generation process is fast, so creating large datasets of these functions is feasible. Improvement of the semantic quality of output functions can be attained by training the model on human-written functions. We leave this to future work.

Earlier we noted that we trained our model on randomly generated functions. One may ask: if we can produce randomly generated functions, then why do we need the Grammar VAE as a generative model? The reason why the Grammar VAE is an important and necessary next step for program synthesis is that, unlike randomly generated functions, functions output by the Grammar VAE take on the semantic characteristics of the training dataset. This is an important quality for many (but not all) applications, and can also tell us a great deal about how humans write code.

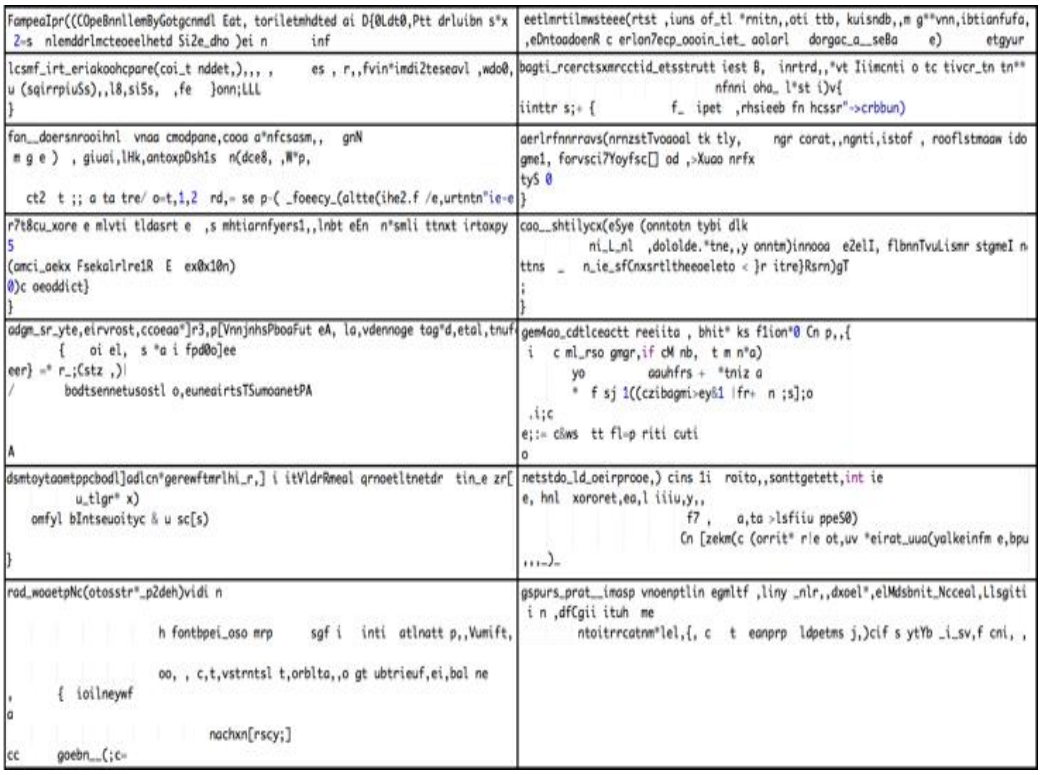


Fig. 4. Character VAE generated functions.

In previous work, a Character VAE has been used as a baseline point of comparison with the Grammar VAE. We follow this convention. We generate 10,000 functions by sampling randomly from the latent space and decoding them, as with the Grammar VAE. The results were much worse. The average function decode time was 0.221 seconds, which is slightly better than the Grammar VAE; however, none of the functions were syntactically or semantically valid. The reason for this poor performance was first speculated by Kusner *et al.* [8], who noted that the Grammar VAE does not need to learn syntax because it automatically produces syntactically valid outputs, whereas the Character VAE must learn both syntax and semantics. Another problem with the Character VAE is that the user has no influence over the basic features of output sequences because there is no analogue to our Grammar VAE constraints subsystem. Fig. 4 shows some of the outputs generated by the Character VAE. Clearly, the Grammar VAE produces higher quality functions, and with very little cost in terms of decode time. We conclude that the Grammar VAE is a good alternative for program synthesis, whereas the Character VAE is not.

4.2. Latent Space

In the following experiments, we determine the coherence of the latent space for both the Grammar VAE and the Character VAE. We say that the latent space is "coherent" if small steps along any dimension in the space produce small or minor changes in functions. Because the latent space is 10-dimensional, we cannot visualize it very well. So, we examine several arbitrarily chosen 2-dimensional cross-sections of the latent space. We do this by sampling a latent vector from the multivariate Gaussian distribution as defined earlier, then choosing 2 dimensions and taking small steps in the positive and negative directions. Finally, we decode all of the latent vectors and place them in a 2-dimensional chart so that regions of the latent space can be visualized. Fig. 5 shows the output of one of these experiments; other experiments had similar results. Not many functions can be visualized at once without making them illegible, so only 5 samples are taken in each direction. In all experiments, we observed that the latent space is coherent, with smooth transitions throughout. At times one can see how a particular dimension controls a particular set of features, as when movement along one axis heavily influences the return type. As a result of these observations, we say that the latent space is coherent. This property is important both because it demonstrates the success of the training procedure, and because it facilitates the search for a function with a particular set of features, which may be necessary depending on the application. Another benefit of a coherent latent space is that we can use it to accurately assess the semantic similarity of functions. We can do this by measuring the distance between latent vectors. Fig. 6 shows an example of latent space distance as a measure of function similarity. In this case, as in many others, latent space distance corresponds very closely with our intuition. This result leads us to another potential application for our work: code plagiarism detection.

The Character VAE did not learn a coherent latent space, making it much less useful for the applications we target. Fig. 7 shows the output of a traversal of the Character VAE's latent space; other experiments had similar results. Again, we conclude that the Grammar VAE is a suitable model for program synthesis, unlike the Character VAE.

Fig. 5. Grammar VAE latent space traversal.

5. Related Work

Gulwani *et al.* [1] give an account of state-of-the-art approaches to program synthesis. In our work, we explore constraint solving and inductive programming in depth. The authors list challenges in the domain, including the intractability of the program space and diversity in user intent. We face both challenges in our

work.

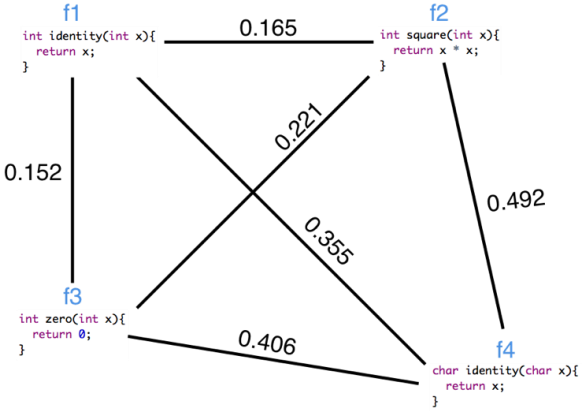


Fig. 6. Grammar VAE latent space distance.

aelwoodd
{
rtu>s (z;
}
	maob2*()
	{
	rotutrn d;;
		cinvac(d
		{
		pturn t;
			mei(oe()
			{
			teiunnw+;;;}
				pdi(v2(f){
				reeurntp0;

fairit()){{ rtrrn 1i;
}
	gai(oc()
	{
	return=0;
		con(arys
		{ eeiugt2 0; }
			foorortd
			{
			return 0;; }
				m3ra .s)
				{
				return ;; }

fwtriy(){
etun (;
}
	c0fne(i) {
	rlturn 00
	}
		m2ynfs(L
		{
		retur8 0; }}
			folotp() {
			return
			;}
				m_inni()
				{
				return 0;;

paonti(){
rtur =;
}
	mvrrbnd)
	{
	erturn ;;}
		melxs(() { eturt 8;;}	_affln()
			{
			aeturn 0; }
				Poadei()){
				netbnn +;

Ottas2(d {
rrturnn ;}}
	_milpde)
	{
	rnturn);
		Matroe)) {
		rtturn 0; }
			mdo(s(d)
			{
			rlturn 0;
			}
				udo(sox(
				{
				eeuunn 0; }

Fig. 7. Character VAE latent space traversal.

Green *et al.* [3], Manna *et al.* [4], and Waldinger *et al.* [5] pioneer early methods of program synthesis. These works rely heavily on a user-defined specification, which is then translated into a proof in a formal logic using SAT/SMT solvers. The proof is used to build the final program. Another approach that relies on user-defined specifications is transformation-based synthesis [12].

Jha *et al.* [13] define formal inductive program synthesizer as one that "generalize[s] from examples by searching a restricted space of programs." Shaw *et al.* [14] design a system that produces recursive LISP programs from single input-output pairs. Summers *et al.* [15] and Biermann *et al.* [16] create more robust LISP programs from multiple input-output examples.

Modern approaches to inductive synthesis often employ machine learning. Liang *et al.* [17] use a Bayesian prior to learn inter-program substructures between related programs using only a few input-output examples. Menon *et al.* [18] learn weights in a probabilistic model using textual features in input-output examples. Both of the aforementioned works rely on a probabilistic context-free grammar that corresponds to a set of function applications. These functions were created by the authors to conduct certain predefined string transformations in a domain-specific syntax. In contrast, our grammar is that of the C programming language, and we allow our model to build functions directly into source code. Balog *et al.* [19] use deep learning to augment traditional inductive programming techniques. Their model, like some of those previously discussed, predicts a series of applications of specific predefined functions (+1, -1, map, reduce, filter, sort, etc.). Bunel *et al.* [20] learn and optimize assembly-like programs using a neural compiler, leading us to consider compiler optimization a potential application.

Some machine learning research has focused on writing source code directly in non domain-specific languages. Lin *et al.* [21] use an RNN and program templates to create single-line bash scripts to solve problems given by a natural language specification. Raychev *et al.* [22] synthesize code completions for programs with holes. They mainly consider Java programs that rely heavily on API calls. Cummins *et al.* [7]

discuss the need for generative models that can produce a large quantity of semantically valid source code for the purpose of training compilers. They note the lack of datasets for such problems, and the need for generated programs to be similar to hand-written programs. They use an RNN to produce code at the character level. Generated code is then fed into a discard filter, which rejects functions that do not compile. They have a discard rate of approximately 32%. We remove the need for a discard filter by employing a Grammar VAE.

Modern inductive approaches to program synthesis have expanded on source code composition by including a grammar to assist in the derivation of generated programs. Patra *et al.* [23] create a generative model for fuzzing similar to Cummins *et al.* [7], except that theirs is based on probabilistic decision trees and a CFG. However, because they do not rely on any context information, only 96.3% of generated JavaScript programs are syntactically valid and 14.4% of the subset of syntactically valid programs execute without causing an error. Our approach guarantees that all generated programs are syntactically valid, and we improve upon these semantic validity numbers.

One final set of sources inspired this work. Most important and relevant to our work is that of Kusner *et al.* [8], who introduce the Grammar VAE neural network architecture and demonstrate its effectiveness on generative modeling of discrete data. They recognize that many generative models across all domains (including program synthesis) often produce invalid outputs. The Grammar VAE guarantees syntactic validity of output sequences when the data can be represented as production rules from a context-free grammar. They find that the Grammar VAE learns a more meaningful latent space when compared to the traditional VAE architecture, and they apply their model to both symbolic regression and molecular synthesis. Finally, they suggest program synthesis as a potential application for the Grammar VAE. Dolan *et al.* [11] design a system to inject vulnerabilities into C source code. They successfully add vulnerabilities to common Unix utilities like bash and tshark. Depending on the application, they are able to inject vulnerabilities successfully between 9.6% and 53.2% of the time. We improve upon these numbers, although we work in source code rather than compiled binaries. Additionally, their vulnerability injection process is quite expensive, taking up to several minutes on average. This latency is unsatisfactory for making a dataset of hundreds of thousands or millions of functions, which is the size of the corpus required by MUSE.

6. Conclusion

Of the models that we implemented, we find that the one with the most use as a generative model for the applications that we target is the Grammar VAE. The Grammar VAE guarantees that every latent vector decodes to a syntactically valid function, over 99% of which can be converted into semantically valid functions using our repair constraint. The importance of this guarantee cannot be overstated. In order to build datasets for our target applications, we need to produce not merely syntactically valid but also semantically valid functions. The Grammar VAE is also fast. As the experiments demonstrate, generating examples from the neural network takes less than a quarter of a second per function on average. This is important because it means that building very large datasets is feasible. Our experiments also show that constraint enforcement works flawlessly for all but the most sophisticated constraints, and even these have error rates below one tenth of one percent. The user therefore has a fair amount of control over the types of functions that appear in the output dataset, and whether or not they contain vulnerabilities. Furthermore, the Grammar VAE contains a very coherent latent space, which gives users the ability to generate datasets of similar functions. The Character VAE, which writes code one character at a time, was our baseline throughout the experiments. We found that this model was not a feasible alternative to the Grammar VAE for our applications. In particular, the Character VAE was not capable of producing syntactically valid C functions, did not allow for constraint enforcement, did not significantly improve execution time, and did not have a coherent latent space. Thus, we

conclude that the Grammar VAE, not the Character VAE, is the best alternative for program synthesis.

This work makes numerous contributions to the domains of both program synthesis and machine learning. First, we provide an end-to-end system that allows users to generate arbitrarily large datasets of syntactically and semantically valid C functions of approximately 2 to 10 lines in length. Second, we make several improvements on the work of Kusner *et al.* [8], who introduce the Grammar VAE. Unlike previous work, our Grammar VAE guarantees that every vector in the latent space will correspond with a syntactically valid output. Another major improvement is the extension of the Grammar VAE into a context-sensitive environment. We use a significantly larger grammar compared to Kusner *et al.* [8], and we can semantically repair almost all output sequences. Third, we demonstrate the ability to exert control over function behavior by imposing constraints on generated functions, including vulnerability injection. Fourth, we improve upon previous methods of program synthesis. Previous generative models for code, like that of Cummins *et al.* [7], often produce invalid outputs—specifically, the aforementioned model must discard at least 32% of all generated functions, and they only consider the domain of OpenCL programs; we can produce syntactically valid C code 100% of the time and semantically valid C code over 99% of the time.

We envision several applications for this work. The first is to generate arbitrarily large datasets of code on which to train machine learning algorithms. This would be useful for projects like MUSE. The second application that we identify is compiler testing and optimization. Our model could be used to create a wide variety of programs for testing or verifying a compiler's correctness; we also believe that our model could be applied to compiler optimization on various tasks. Cummins *et al.* [7] use an RNN to synthesize OpenCL programs to train their compiler, which leveraged machine learning techniques to improve its performance on certain tasks. They note that there is an increasing need for datasets of code that finely cover the feature space of programs. We can fill this gap. Finally, we believe that our model could be a valuable addition to source code plagiarism detection tools or other applications that require a similarity metric between pieces of code. We propose a new metric to these tools: the distance between the latent vectors associated with two functions. Because of the coherence of the latent space, this distance would give a good measure of how similar two functions are from a qualitative standpoint.

Acknowledgment

This project was sponsored by the Air Force Research Laboratory (AFRL) as part of the DARPA MUSE program. It was also supported by the Draper Fellows program.

References

[1] Gulwani, S., Polozov, O., Singh, R., *et al.* (2017). Program synthesis. *Foundations and Trends® in Programming Languages*, *4*, 1-2, 1–119.

[2] Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2017). A survey of machine learning for big code and naturalness. arXiv preprint arXiv:1709.06182.

[3] Green, C. (1969). Application of theorem proving to problem solving. *Proceedings of the 1st International Joint Conference on Artificial Intelligence (IJCAI'69).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[4] Manna, Z., & Waldinger, R. J. (1971). Toward automatic program synthesis. *Commun.*, *3*, 151–165.

[5] Waldinger, R. J., & Lee, R. C. T. (1969). PROW: A step toward automatic program writing. *Proceedings of the 1st International Joint Conference on Artificial Intelligence (IJCAI'69)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA,

[6] Manna, Z., & Waldinger, R. (1975). Knowledge and reasoning in program synthesis. *Artificial Intelligence 6*(*2)*, 175-208.

[7] Cummins, C., Petoumenos, P., Wang, Z., & Leather, H. (2017). Synthesizing benchmarks for predictive modeling. *Proceedings of the 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

[8] Kusner, M. J., Paige, B., & Hernández-Lobato, J. M. (2017). Grammar variational autoencoder. arXiv preprint arXiv:1703.01925.

[9] Weininger, D. (1988). SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules. *Journal of Chemical Information and Computer Sciences*, *28(1)*, 31–36.

[10] Kernighan, B., & Ritchie, D. M. (2017). *The C Programming Language*. Prentice hall.

[11] Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F., & Whelan, R. (2016). Lava: Large-scale automated vulnerability addition. *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP)*.

[12] Manna, Z., & Waldinger, R. (1974). Knowledge and reasoning in program synthesis. *IBM Germany Scientific Symposium Series*. 236–277, Springer.

[13] Jha, S., & Seshia, S. A. (2017). A theory of formal synthesis via inductive learning. *Acta Informatica*. 1–34.

[14] Shaw, D., Wartout, W., & Green, C. (1975). Inferring LISP programs from examples. *IJCAI*, *75*, 260–267.

[15] Summers, P. D. (1986). A methodology for LISP program construction from examples. *Readings in Artificial Intelligence and Software Engineering*, 309–316.

[16] Biermann, A. W. (1978). The inference of regular LISP programs from examples. *IEEE transactions on Systems, Man, and Cybernetics*, *8*, 585–600.

[17] Liang, P., Jordan, M. I., & Klein, D. (2010). Learning programs: A hierarchical Bayesian approach. *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*.

[18] Menon, A., Tamuz, O., Gulwani, S., Lampson, B., & Kalai, A. (2013). A machine learning framework for programming by example. *Proceedings of the International Conference on Machine Learning* (pp. 187–195).

[19] Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., & Tarlow, D. (2016). Deepcoder: Learning to write programs. arXiv preprint arXiv:1611.01989.

[20] Bunel, R. R., Desmaison, A., Mudigonda, P. K., Kohli, P., & Torr, P. (2016). Adaptive neural compilation. *Advances in Neural Information Processing Systems*, 1444–1452.

[21] Lin, X. V., Wang, C., Pang, D., Vu, K., & Ernst, M. D. (2017). *Program Synthesis from Natural Language using Recurrent Neural Networks*. USA.

[22] Raychev, V., Vechev, M., & Yahav, E. (2014). Code completion with statistical language models. *ACM SIGPLAN Notices*, *49*, 419–428.

[23] Patra, J., & Pradel, M. (2016). Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data. TU Darmstadt, Department of Computer Science, Tech. Rep.

Leonard Kosta was born in Elizabeth city, North Carolina in 1995. He earned a BS in computer science and Russian from the United States Military Academy at West Point, New York in 2017. His undergraduate research was in high performance computing systems and data science.

He is currently serving in the U.S. Army as a cyber warfare officer and conducts research as a draper fellow in Cambridge, Massachusetts while attending graduate school at Boston University. He will earn his MS in computer science in 2019. Before working at Draper, he also interned at the U.S. Army Engineer Research and Development Center, performing research on high performance computing systems that resulted in the publication of "Measuring I/O Performance of Lustre

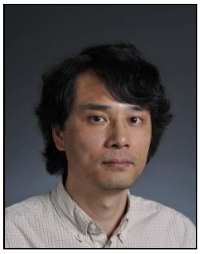
and the Temporary File System for Tradespace Applications on HPC Systems." His current research is in the use of tracking algorithms and machine learning to track and classify ships using sensor data from floats.

First Lieutenant Kosta was awarded the U.S. Grant Memorial Award and the Colonel Phillip Matthews Award for excellence in computer science and Russian, respectively. He was also awarded the Major General & Mrs. Russ Fuhrman Scholarship by the Society of American Military Engineers for outstanding performance as a Cadet at West Point.

Laura Seaman earned a Ph.D in bioinformatics and the M.A in statistics from the University of Michigan in Ann Arbor Michigan in 2017. She received a M.S in biological engineering from the Massachusetts Institute of Technology in Cambridge Massachusetts in 2013. Her graduate work on "The 4D Nucleome of Cancer" used graph analytics methods to study the structure of DNA and the changes that occur during cancer.

She currently works as a senior machine intelligence scientist at Draper in Cambridge Massachusetts. Before starting graduate school, she interned with the national nanotechnology infrastructure network at Georgia tech, worked in the Robert Langer lab on targeted nanoparticle drug delivery, and intern at Biogen building a machine learning algorithm to predict part failures during quality control. Her current work focuses on applying machine learning techniques to a variety of fields including neural image processing of biological tissue and neural networks and natural language processing for financial applications.

Dr. Seaman was awarded the outstanding contributed talk at controlling complex networks, NetSci 2017, and EDGE award for graduate education, and a Rackham merit fellowship in 2013.



Hongwei Xi received his PhD in pure and applied logic from Carnegie Mellon University in 1998. Since 2007, he has been appointed associate professor in the Department of Computer Science, Boston University.

His main research applies advanced type theory to programming language design and implementation. He is currently the primary designer, implementer and maintainer of ATS, a functional programming language equipped with linear and dependent types that formally combines program specification with program implementation. He has also done a large body of pioneering work on supporting dependent types and linear types for practical programming.

Dr. Xi served on the program committees of several prestigious ACM conferences on programming languages (e.g., POPL, PLDI and ICFP).