Generating Vulnerable Code via Learning-Based Program Transformations

Haipeng Cai and Yu Nong and Yuzhe Ou and Feng Chen

Abstract Software vulnerabilities are a major source of cybersecurity threats. Therefore, it is of paramount importance to defend against (e.g., detect, repair) them. Data-driven approaches, especially those based on machine/deep learning (ML/DL), have demonstrated a great potential to that end. To achieve practical efficacy, these approaches rely on a large number of training samples. However, currently such samples, especially those that are known as vulnerable, are not richly available, immediately impeding ML/DL applications for software vulnerability analysis. Moreover, these samples would also meet the critical need for making scientific progress in software assurance through objective benchmarking of existing techniques and tools.
In this chapter, we describe a learning-based approach to generating vulnerable code samples, so as to empower both the scientific assessment of extant software security defense solutions and the development of new ones. We formulate the sample generation problem as that of learning the common patterns of code changes that introduce vulnerabilities in existing (seed) samples, followed by applying such changes to given clean programs. We also present our empirical results that show the promise and discuss the gaps with our approach, while examining several key factors in the design of effective DL-based sample generation.

Haipeng Cai
Washington State University, Pullman, WA 99163, e-mail: `haipeng.cai@wsu.edu`

Yu Nong
Washington State University, Pullman, WA, 99163, e-mail: `yu.nong@wsu.edu`

Yuzhe Ou
University of Texas at Dallas, Richardson, TX 75080, e-mail: `yuzhe.ou@utdallas.edu`

Feng Chen
University of Texas at Dallas, Richardson, TX 75080, e-mail: `feng.chen@utdallas.edu`

1 Introduction

The largely varied vulnerabilities in software programs are a primary cause of security threats in the cyber world. Indeed, the pervasive presence of these code vulnerabilities has constituted a grand challenge to software assurance. In response, researchers have proposed and continue to propose numerous defense (e.g., prevention, detection, and repair) and countermeasure techniques (e.g., [5, 3, 12, 4]) to defend against particular types of vulnerabilities.

However, there is a critical lack of objective measurements and scientific comparisons of these techniques [9], which immensely impedes the fundamental advancement in software security assurance. To support comparative assessments of the current state-of-the-art techniques, a large set of testing benchmarks with vulnerability ground truth is required, which is also currently lacking. In addition, the development of powerful data-driven techniques (e.g., machine/deep learning-based vulnerability detectors) heavily relies on the availability of a large number of samples (for training, validation purposes). Yet the sizes of existing vulnerability datasets are generally small. A large set of samples that represent real-world vulnerabilities with a high coverage of various vulnerability types is not available. Many such datasets that are currently available also do not come with detailed ground truth (i.e., what the true-positive vulnerabilities are in a benchmark) [10], which constitutes another major barrier for effective technical development and assessment.

To fill this current gap, in this chapter, we describe a deep learning (DL) based approach that automatically generates vulnerable samples. We focus on generating vulnerable samples because normal/clean ones are already richly available. Our approach also takes advantages of this rich availability, producing vulnerable code by injecting vulnerability-introducing code changes to normal/clean programs. Inspired by existing neural code editing techniques, we combine graph neural networks (GNN) to learn the representations of input program pairs (a vulnerable program and its vulnerability-fixed version), with a sequence model to predict code changes that inject vulnerabilities. In particular, from the given pairs of programs as training samples, our neural model learns the code location and edit actions for the code changes between each pair. The training process is supervised by an objective that aims to minimize the differences between the vulnerable programs generated from the corresponding normal versions and the ground-truth vulnerable versions in the training set. Then, the trained model is used to generate the vulnerable versions for any given normal programs, realizing vulnerability data augmentation. While the edits are made on the abstract syntax tree (AST) of each input (normal) program, the predictions on what edits to make and where are based on the semantic encoding of the program via the GNN, which learns the complex structures of the code.

To empirically evaluate our approach, we implemented it by adapting Graph2Edit [13], a state-of-the-art neural code editor for bug repair. Then, we took a well-known, existing vulnerability benchmark suite SARD [1], using part of it for training and the rest for testing. We measure the effectiveness of our approach for

[1] https://samate.nist.gov/SARD/

vulnerability injection in terms of recall with respect to the ground truth. Our results show that generating vulnerable samples by learning to inject vulnerability-introducing changes has promising merits, despite several limitations concerning some requirements for the input programs. We also gauged the efficiency of our approach and therefore demonstrated its practicality in terms of execution time and memory costs.

In the rest of this chapter, we first elaborate the existing dataset (§2) that we used for developing our DL model, followed by describing the technical approach (§3). We provide important tool implementation details in §4. Then, we present our evaluation experiments and empirical results in §5. Based on these results, we discuss lessons learned while providing insights for future development of learning-based vulnerability data augmentation (§6) before making a brief concluding remark (§7).

2 Mining Existing Vulnerability Data

We started by mining existing vulnerability data used in the literature. We will use this dataset to develop and validate our technical approach.

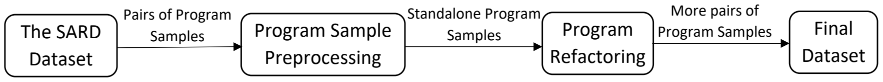
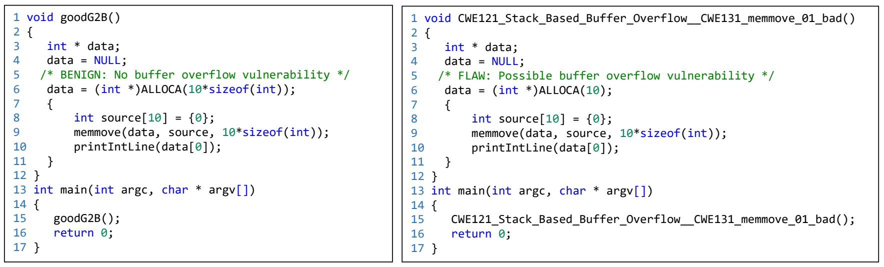

The SARD Dataset
Pairs of Program Samples
Program Sample Preprocessing
Standalone Program Samples
Program Refactoring
More pairs of Program Samples
Final Dataset

Fig. 1: The dataset mining/processing work flow on SARD-Buf.



```
void goodG2B()
{
   int * data;
   data = NULL;
  /* BENIGN: No buffer overflow vulnerability */
   data = (int *)ALLOCA(10*sizeof(int));
   {
       int source[10] = {0};
       memmove(data, source, 10*sizeof(int));
       printIntLine(data[0]);
   }
}
int main(int argc, char * argv[])
{
    goodG2B();
    return 0;
}
```

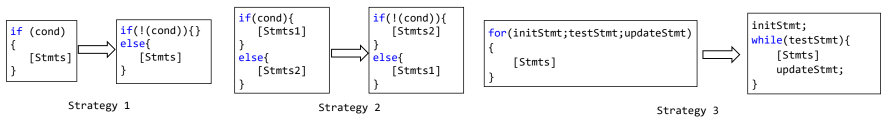
```
void CWE121_Stack_Based_Buffer_Overflow__CWE131_memmove_01_bad()
{
   int * data;
   data = NULL;
  /* FLAW: Possible buffer overflow vulnerability */
   data = (int *)ALLOCA(10);
   {
       int source[10] = {0};
       memmove(data, source, 10*sizeof(int));
       printIntLine(data[0]);
   }
}
int main(int argc, char * argv[])
{
    CWE121_Stack_Based_Buffer_Overflow__CWE131_memmove_01_bad();
    return 0;
}
```

Fig. 2: An example sample in *SARD-Buf* (left: clean code; right: vulnerable version).

The dataset we looked into was SARD [2], widely used by prior research on code vulnerability detection/repair. Figure 1 shows the overview of mining this dataset,

[2] https://samate.nist.gov/SARD/

which includes pairs of vulnerable program samples of different vulnerability types and corresponding fixed (non-vulnerable/clean) versions. We begun with the 6,540 pairs of program samples involving buffer-overflow vulnerabilities and refer to this subset as *SARD-Buf*. We then preprocessed these program samples, including merging files that together represent a holistic functional unit and eliminating unnecessary macros, so that each program sample was stand-alone and can be compiled into an executable. Figure 2 shows an example of these sample pairs. The sample on the left is the clean/non-vulnerable program in the pair, while the sample on the right is the respective vulnerable program, where a buffer-overflow vulnerability is rooted at line 6 and manifested at line 9.


```
if (cond)
{
    [Stmts]
}
```
```
if(!(cond)){}
else{
    [Stmts]
}
```

```
if(cond){
    [Stmts1]
}
else{
    [Stmts2]
}
```
```
if(!(cond)){
    [Stmts2]
}
else{
    [Stmts1]
}
```

```
for(initStmt;testStmt;updateStmt)
{
    [Stmts]
}
```
```
initStmt;
while(testStmt){
    [Stmts]
    updateStmt;
}
```

Fig. 3: Refactoring transformations considered so far.

To suit our work, we refactored the original samples in order to diversify them, through three kinds of refactoring transformations shown in Figure 3. The first two negate the conditions of the `if` or `if-else` statements and accordingly swap the `then` and `else` blocks, while the third converts `for` loops into `while` loops. The refactoring did not change the semantics of the programs, but the code structure changed, resulting in additional structural diversity of the dataset. A program may have more than one locations that can be refactored. In this case, we refactored multiple locations in a combinatorial manner. For example, if a program has n locations suitable for refactoring, the total number of refactored programs would be $2^n - 1$. Since the number could be very large when n is large, we limited the number of refactored programs for each original program to 101. Finally, we obtained a dataset containing a total of 32,654 pairs of program samples.

Our characterization revealed that, in *SARD-Buf*, most samples had less than 150 lines of code and 5 functions. The differences between the non-vulnerable and vulnerable samples in most of the pairs were less than two lines of code. Also, most of the edits between a pair are limited to one code line. The unique edit patterns are very few (less than 10). Given the excessive simplicity of *SARD-Buf*, we proceeded with mining the entire set of C samples in SARD, referred to as *SARD-Full*. This dataset includes 38,802 original (i.e., without refactoring) pairs of programs, covering 109 vulnerability types. To avoid introducing duplication, we did not perform refactoring on the *SARD-Full* dataset.

Table 1 lists the vocabulary size of each of the datasets mentioned above, treating the code as sequences of tokens each as a word. Further, Table 2 gives the percentages of samples in each datasets that have various ranges of code change size. These numbers offer a view of key characteristics of these datasets that are highly relevant to our tasks

Table 1: The size of vocabulary

Dataset	*SARD-Buf*	*SARD-Full*
Size	580	3,193

Table 2: Sample (percentage) distribution by the number of lines modified between each pair

Dataset	**1**	**2-5**	**6-10**	**11-20**	**21-50**	**>50**
SARD-Buf	77.50%	19.82%	0.23%	0.82%	0.26%	1.29%
SARD-Full	35.60%	29.03%	17.45%	7.84%	2.58%	7.29%

on training deep learning models for vulnerable code generation via vulnerability injection (i.e., injecting vulnerability-introducing changes to the non-vulnerable version in each pair).

3 Learning-Based Data Generation via Vulnerability Injection

Among other applications, recent advances in deep learning (DL) have enabled cutting-edge results not only in classification tasks (e.g., recognizing cars in autonomous-driving scenes) but also for generating novel data samples (e.g., producing a variety of human faces from a given input face image). In particular, applications of these promising capabilities in the software systems domain emerge rapidly as well, ranging from automated generation of software documentation to translating code in one programming language to others. Especially promising are latest progresses in building deep code models to automatically fix software bugs. For example, neural machine translation (NMT) models are used for automatic program repair [6, 7] by treating code as natural language token sequences. For the same or similar purposes, in [13, 2, 1, 14], graphs neural networks (GNNs) are used to learn program syntax and code structure hence predict bug-fixing code changes or bug-fixed code directly. However, to the best of our knowledge, there has been no research looking into leveraging similar power of DL for software vulnerability injection.

While conceptually promising for vulnerability data generation/augmentation, automatically injecting vulnerabilities into clean programs faces several technical challenges that have not been studied or addressed:

- Unlike in the traditional DL target domains (images, natural-language texts, videos), software programs have rich semantics in addition to strong syntactic constraints; as a result, small changes to a program can lead to major changes in the program's semantics and/or syntactic violations. The former could make the changed program malfunction unexpectedly at runtime, while the latter can result in the changed code not even being compilable (i.e., invalid code).
- Although DL has the advantage of avoiding the tedious and unscalable manual feature engineering process, useful DL models are typically trained on hundreds

of thousands or even millions of samples (e.g., the widely used ImageNet dataset contains 14 millions of sample images); such sizable datasets for software vulnerability analysis are not available. In fact, if there are, we would not even need to solve the problem itself (i.e., injecting vulnerabilities into clean programs so as to generate/augment vulnerable program samples).

Inspired by Graph2Edit [13], a state-of-the-art DL-based bug repair approach using neural code editing, we propose to overcome the above challenges through an encoder-decoder model. In particular, the encoder learns program representations through graph neural network (GNN) modeling, which not only learns syntactic code tokens in given programs but also encodes the relationships among all the code entities in each program sample. Based on the learned representation, the decoder then learns to transform the inputs programs by predicting code edits rather than generating the target code directly. Importantly, like Graph2Edit, we decompose every code editing step in a way that emulates how a human developer commonly thinks about composing a code change: (1) decide the type of code change (i.e., operator type), such as adding or deleting a code token; (2) locate where to make the change (i.e., the code entity to change); and (3) determine the value required (e.g., operands in an expression) in some types of changes (e.g., adding code).

3.1 Problem Formulation

Typically, program bug repair is formulated as the problem of gradually making code edits (changes) in order to transform the given buggy program into its fixed (bug-free) version. In other words, the process is one of applying bug-fixing changes to the input buggy program. We formulate *vulnerability injection* as a reversed program transformation: applying bug (i.e., vulnerability)-introducing code changes to the input bug-fixed (i.e., vulnerability-fixed, or clean) program.

To solve this problem using deep neural networks, we propose to learn a probabilistic model that iteratively predicts a sequence of code edits (i.e., editing operations) that transforms the input clean code into the target code (i.e., the vulnerable version of the input code). Instead of directly editing the text of code, the operations are performed on a tree representation of the code. For this representation, we use abstract syntax trees (ASTs), which capture the syntactic structure of source code. The rationale is that by doing so, we have an immediate control of the syntactic validity of the resulting (i.e., edited) code by enforcing syntax with respect to the targeted programming language during the decision making (predictions) by the neural networks.

In particular, we adopt the three-level hierarchy in formulating each edit (editing operation) proposed in [13], corresponding to the three elements of each editing step outlined above: operator, edit location, and value. Accordingly, the problem of predicting an edit is further formulated as (1) $\underline{\text{selecting the operator}}$, (2) $\underline{\text{selecting the (AST) node to edit}}$, and (3) $\underline{\text{choosing value when necessary}}$, in that order, noted as edit operation *primitives*. Then, a model that accurately predicts the needed (AST) tree operations to transform the source AST into the target one is learned by minimizing the classification loss under the supervision of the (ground-truth) edits in the training samples.

3.2 Model Design

Given a set of program pairs $\{\langle C_-^{(i)}, C_+^{(i)}\rangle\}_{i=1}^N$, where each pair $\langle C_-, C_+\rangle$ contains the input clean code (original/before-change program) C_- and the target vulnerable code (changed/vulnerability-injected program) C_+. We aim to train a model that can predict the T incremental tree edits, denoted as $a_{1:t} := (a_1, a_2, ..., a_t)$ (e.g. deleting or adding an AST node), that transform AST(C_-) to AST(C_+).

The model to predict the edit sequence involves training an edit encoder function f_Δ, which takes $\langle C_-, C_+\rangle$ as input and output an vector encoding of edit representation $f_\Delta(C_-, C_+) \in \mathbb{R}^n$. Let $g_{1:t} = (g_1, ..., g_t)$ denote the tree history of applying an edit sequence $a_{1:T}$ to C_-, where g_1 is the AST of C_- and $g_{\tau+1}$ is the resultant AST at time step τ when applying the edit a_t to g_τ, the edit sequence prediction is then described as the following autoregressive process:

$$\begin{aligned} p(a_{1:T} | f_\Delta(C_-, C_+), C_-) &\equiv p(a_{1:T} | f_\Delta(C_-, C_+), g_1) \\ &= \prod_{t=1}^{T} p(a_t | f_\Delta(C_-, C_+), g_{1:t}) \end{aligned} \tag{1}$$

where g_1 is determined by the source code (in the form of AST) of the input program C_-, g_{t+1} is determined by g_t and a_t.

As mentioned earlier, to predict an edit operation a_t to take, we need to determine its three lower-level primitives: the operator type op_t, the operated node location loc_t, and the optional value specification val_t for adding a node. As considered in [13], for each edit operation a_t we consider four possible operator types: (1) *delete a node*, (2) *add a node*, (3) *copy a sub-tree*, and (4) *STOP editing*. Thus, $op_t \in$ {`Delete`, `Add`, `CopySubTree`, `Stop`}. In this way, each action a_t itself can be modeled as a sequence of primitives $a_t = (op_t, loc_t, val_t)$.

To track the tree history information, an encoding s_t which serves as the hidden state is introduced and calculated via an LSTM encoder, i.e $\boldsymbol{s}_t = \text{LSTM}([g_t; f_\Delta], \boldsymbol{s}_{t-1})$. The prediction of edit operation a_t at current time step t is then summarized as:

$$p(a_t | s_t, g_t) = p(op_t | \boldsymbol{s}_t) p(loc_t | \boldsymbol{s}_t, g_t, op_t) p(val_t | \boldsymbol{s}_t, g_t, op_t, loc_t) \tag{2}$$

where the conditionals express the following three prediction steps in the holistic prediction of each edit operation.

Selecting the operator. This step is a 4-class classification problem. The probability of taking an operator op_t is calculated through a multilayer perceptron (MLP): $p(op_t | \boldsymbol{s}_t) = \text{softmax}(\boldsymbol{W}_{op} \boldsymbol{s}_t + \boldsymbol{b}_{op})$.

Selecting the (AST) node to edit. Since at each timestamp t, the AST tree (g_t) can contain an arbitrary number of nodes, there are multiple choices of node that we can potentially operate on. To select a target node, we learn a hidden state $\boldsymbol{h}_{node,t} = tanh(\boldsymbol{W}_{node}[\boldsymbol{s}_t; \text{emb}(op_t)] + \boldsymbol{b}_{node})$ as the "pointer", where emb$(\cdot)$ is a learnable embedding function. For a tree with S candidate nodes that op_t can operate on, the probability of loc_t taking particular candidate node locations is $P(\cdot|s_t, op_t) = softmax\big[(\langle \boldsymbol{h}_{node,t}, \boldsymbol{n}_{t,i}\rangle)_{i=1}^{S}\big]$, where $\boldsymbol{n}_t = (\boldsymbol{n}_{t,1}, ..., \boldsymbol{n}_{t,S})$, which are the node representations of g_t extracted by a gated graph neural network (GGNN [8]).

Choosing value when necessary. In this step, the model aims to predict an argument val_t for the `Add` and `CopySubTree` actions if one of these two is predicted in the first (i.e., "Selecting the operator") step; for other operator types, the value is set to be `NULL`. Similar to the predictor for "Selecting the (AST) node to edit", the distribution $P(val_t|\cdot)$ is given by a pointer network, with the hidden state defined as $\boldsymbol{h}_{val,t} = tanh(\boldsymbol{W}_{val}[\boldsymbol{s}_t; \boldsymbol{n}_{t,loc_t}; emb(p_{n_{t,loc_t}} \mapsto n_{t,loc_t}))] + \boldsymbol{b}_{val})$, where $emb(p_{n_{t,loc_t}} \mapsto n_{t,loc_t})$ is the embedding of the type of the edge between the parent node $p_{n_{t,loc_t}}$ and the child node n_{t,loc_t}. The probability of taking some val_t is then given by $P(\cdot|s_t, op_t, loc_t) = softmax\big[(\langle \boldsymbol{h}_{val_{t,j}}, val_{t,j}\rangle)_{j=1}^{V}\big]$, where V is the dictionary size for the value.

Since the edit sequence can be viewed as a sequence of sequence (of primitives), i.e. $a_{1:t} = \{(op_\tau, loc_\tau, val_\tau)\}_{\tau=1}^{t}$, the joint probability of an entire edit sequence is then factorized on the primitive-level conditionals given earlier:

$$\begin{aligned} p(a_{1:t}|f_\Delta, g_1) &= p(a_{1:t}|s_1, g_1) \\ &= p(op_1|s_1)p(loc_1|s_1, g_1, op_1)p(val_1|s_1, g_1, op_1, loc_1)... \\ &\quad p(op_t|s_t)p(n_t|s_t, g_t, op_t)p(val_t|s_t, g_t, op_t, loc_t) \end{aligned} \quad (3)$$

where s_1 is the initial state and $\boldsymbol{s}_t = \text{LSTM}([g_t; f_\Delta(C_-, C_+)], s_{t-1})$. In the above decomposition, g_{t+1} is determined by g_t and $a_t = (op_t, loc_t, val_t)$, where performing the tree operation a_t on the tree g_t yields the new tree g_{t+1} for the next time step of the AST being edited.

3.3 Model Training and Testing

In the training stage, the input dataset (i.e., training set) is a set of paired code samples $\langle C_-, C_+\rangle$, which is the vulnerability-fixed and vulnerable version of a program, respectively. With these paired samples, the training algorithm aims to compute the model parameters to maximize the probability of taking the "right edits". This process needs the supervision of the gold-standard sequence that transforms C_- to C_+ in each pair. Note that there may be a number of sequences that would perform the same transformation on the code. We follow the approach in [13] to address this ambiguity by finding the shortest-length tree edit sequence to transform the AST of C_- to that of C_+, denoted as $a^*_{1:T}$, using a dynamic programming algorithm. Then, the training algorithm is essentially to solve the optimization problem to minimize the expected negative log-likelihood:

$$\min_{\theta} \mathcal{L}(\theta) := \mathop{\mathbb{E}}_{(C_-,C_+)\sim\mathcal{D}} -\log p(a^*_{1:T}|f_\Delta(C_-,C_+),C_-,\theta) \tag{4}$$

At testing (inference) time, C_+ is missing; given a C_-, the learned model takes another similar pair of code $\langle C'_-, C'_+\rangle$ in the training set and extracts the edit representation to predict edits on C_-. The trained model selects the action sequence that maximizes the probability $p(a_{1:t}|f_\Delta(C'_-,C'_+),g_1;\theta)$ through a beam search. Once the special *STOP* operator is predicted at time step T, the final edit sequence $a_{1:T}$ is formed. This final edit sequence leads to the edited version of the input program.

4 Technique Implementation

The correctness of our vulnerability-injection technique relies on the syntactic validity of the edited (i.e., output) code, which is determined by that of the generated AST for the given input program (also in the form of its AST). The AST syntax is defined and described using what is called the *Abstract Syntax Description Language* (ASDL) [15]. The current tool that implements Graph2Edit [13] only works with C# programs, accordingly its ASDL implementation also targeted C#. We aim at generating C vulnerable samples in terms of tool implementation—the seed vulnerability dataset (SARD) that we chose to use is also a set of (paired) C programs. Thus, we first developed our own ASDL for C and accordingly the AST generator for a given C program that conforms to the ASDL for C.

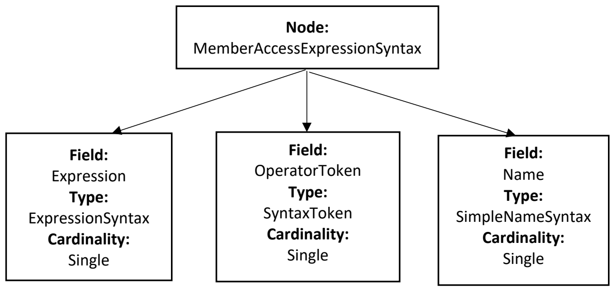


Fig. 4: An illustration of a node in the tree representation of ASDL.

Figure 4 shows an example of an ASDL node. In each ASDL node, there are one or more fields indicating the necessary information needed in that kind of nodes. In the example, a `MemberAccessExpressionSyntax` has three fields: `Expression`, `OperatorToken`, and `Name`. The content in each field must follow the type (i.e., `ExpressionSyntax`, `SyntaxToken`, or `SimpleNameSyntax`). Cardinality indicates whether the content in that field is *single*, *optional*, or *multiple*.

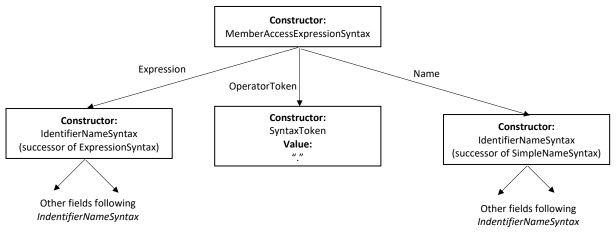


Fig. 5: An example AST following the syntax defined by the corresponding ASDL.

The ASTs of the program samples to be accepted by our DL model, as well as edits on the ASTs, must follow the syntax defined by the ASDL. Figure 5 shows an example of AST that follows the syntax in Figure 4. A member access expression (i.e., `Obj1.Obj2`) is parsed into an AST. "`Obj1`" is in `IdentifierNameSyntax` which is a successor of `ExpressionSyntax`. "`.`" is a `SyntaxToken`. "`Obj2`" is another `IdentifierNameSyntax` which is the successor of `SimpleNameSyntax`. The `IdentifierNameSyntax` node has other fields which follow its syntax defined.

Overall, there are 70 nodes in the ASDL for C that we developed, as illustrated in Figure 6. The other parts of our technique were implemented by adapting respective ones in the Graph2Edit implementation in Python based on PyTorch [11].

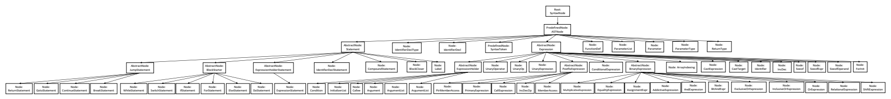


Fig. 6: Our abstract syntax description language (ASDL) developed for C.

5 Empirical Evaluation of Performance

For a preliminary evaluation of the effectiveness of the described technique in generating vulnerable code from their clean versions, we conducted empirical studies guided by the following research questions (RQs):

? Questions

RQ1: *How effective is our technique for generating vulnerable code?*

Given a dataset with pairs of clean (non-vulnerable) and vulnerable program samples, we randomly split these pairs into training and testing sets. We train our DL model using the training set. Then, we measure the effectiveness of our technique in terms of the percentage of ground-truth vulnerable programs that it was able to generate from the given clean versions of those samples as inputs (i.e., using the recall as the main effectiveness metric).

RQ2: *How efficient is our technique for vulnerable code generation?*

Efficiency is important for a DL-based approach's practicality. A model that cannot scale to a sizable dataset would hit scalability hence adoptability barriers in practice. Thus, via this question, we evaluate how much time/memory costs our approach incurs for training the DL model.

5.1 Experimental Setup

Before presenting the evaluation results, we describe how we obtained them, elaborating in particular the *data configuration* and evaluation *metrics* used.

Data configuration: To evaluate effectiveness, we used the *SARD-Full* dataset (38,802 pairs of programs) to train, validate, and test the tools. We split it into training, validation, and testing set with a ratio 80% (31,042), 10% (3,881), and 10% (3,879), respectively. Some of the program pairs cannot be processed by our tool. Thus, we only used the pairs that can be processed successfully. Specifically, the tool implementation encountered out-of-memory issues against the pairs related to "CWE78" and "socket". Thus, we left out those pairs and hence had 22,060 pairs for training and 2,755 pairs for validation.

Metrics: The *SARD-Full* dataset includes samples that are all paired (i.e., one vulnerable program along with its clean version). And we used the clean versions as inputs during the testing of our model. If the target program generated by the model exactly matches the ground-truth vulnerable version of the program in terms of their ASTs, the testing case is marked as a true positive (TP)—that is, the expected output is vulnerable and the actual output is vulnerable also; otherwise, it is marked as a false negative (FP)—that is, the expected output is vulnerable but the actual output is not considered vulnerable. We note that this current measurement scheme is not perfect because when the model-generated program $S'_{vulnerable}$ for an input program S_{clean} does not match the ground-truth vulnerable version of S_{clean}, it does not necessarily mean that $S'_{vulnerable}$ is surely not vulnerable. Yet manually checking these non-exactly-matched outputs can be quite tedious given the size of our dataset.

More importantly, we do not expect our current model to have a great chance of injecting vulnerabilities in a “novel" manner while producing a valid program. For these reasons, we chose to use this imperfect way of computing false negatives.

However, with respect to this ground truth and the problem setting of vulnerable sample generation via vulnerability injection, we could not compute false positives and true negatives, because in our ground truth we do not have any S_{clean} paired with a non-vulnerable version to compare the model output against. As a result, we only compute $Recall = \frac{\#TP}{\#TP+\#FN}$, and used it as our main effectiveness metric.

5.2 Evaluation Results

In this section, we present and interpret our empirical results on the effectiveness and efficiency of the proposed approach.

RQ1: Effectiveness. In terms of effectiveness (recall), although the tool cannot process all the pairs in the original dataset for training because of the issue explained above, it was able to process all the pairs for testing since the testing phase costed much less time and memory because of the much smaller size of the testing set (hence suffered no out-of-memory issues).

Using aforementioned data configuration, the recall achieved by the tool was 76.12%. As a first step, we believe the described technique is useful and promising based on this result. Meanwhile, we have to note that we have not addressed all of the challenges discussed earlier to vulnerability injection. While not completely trivial, the *SARD-Full* dataset contains paired program samples that are mostly simple in terms of code structure and logic complexity; moreover, the changes between the pairs are generally simple—typically limited to one or two lines (see Table 2), and the highly similar changes appeared in a number of pairs. In addition, compared to those that are dealt with by many DL models in other (e.g., natural language processing) domains, the size of vocabulary of our dataset is relatively small (see Table 1). These characteristics of the dataset make learning the code-change patterns relatively easy, compared to, say, learning from real-world, complex code samples. Thus, evaluating on those samples and different datasets is our immediate next step.

RQ2: Efficiency. For efficiency results, we recorded the total time for training our DL model (including that for preprocessing, representation learning, and batched training). We then computed the average time cost per training sample. We also recorded the highest memory usage in the training process.

On the Nvidia RTX 3090 GPU platform we used, the training was very fast. For each complete epoch, the time cost was about 30 minutes. Memory costs were relatively high: at our chosen batch size of 4, the training costed over 23.4G video memory and 266G physical memory. However, considering modern computing resources and the generally computation-intensive nature of DL models, these numbers are expected to be acceptable, especially given that vulnerable code generation is not an user-interactive task—the computation would not interrupt users' ordinary workflow.

6 Discussion

Based on our current technical design and the results of our preliminary evaluation, we now discuss about the general methodology of using deep learning to generate vulnerable code via program transformations in two dimensions: the key *data characteristics* that may influence the DL model performance, and the *technical limitations* of our approach.

6.1 Data Characteristics

In the encoder-decoder design of our DL model, the encoder focuses on learning the representation of programs based on their ASTs through graph modeling (i.e., using GGNN), while the decoder learns to predict edit sequences through sequence modeling (i.e., using LSTM). Thus, among others, three aspects of characteristics of the data (i.e., program samples) intuitively affect how effective the learning and prediction may be, as discussed below.

Edit pattern frequency. At a high level, the process of model training and inference is that of learning the code-change (i.e., changes that introduce vulnerabilities) patterns in the training samples (during the training phase) and applying the patterns learned to the input clean code (during the inference phase). Therefore, intuitively, how frequent such patterns are instantiated in the training set has an immediate impact on the training and inference performance. It is common knowledge that the batch gradient method (which is used in our model training) favors more repeated patterns, as they weigh more in determination of the gradient direction towards reducing the loss on all training samples in the current batch. The task of vulnerability injection faces a great challenge as the potential input space of "all possible programs" is huge, and the possible combinations of different code structures and feasible injection points are tremendous in number, while subject to constraints of the language syntax. Thus, to get a good estimation with small overall (global) errors, more instances (examples) of the same pattern would then facilitate the model to learn the patterns. Meanwhile, when a pattern has a higher frequency in the training set hence the model learns more stable knowledge about the pattern, it is more likely to succeed in applying the pattern to an arbitrary clean code, hence better model inference performance as well.

Edit sequence length. A single edit that suffices for program transformations to inject a vulnerability is rare; more typically, the program transformations correspond to a sequence of edits. The length of an edit sequence is then straightforwardly the number of edit operations in the sequence. This length determines the number of time steps required for the autoregressive decoder to make a correct prediction. Also, the greater the edit sequence length is, the harder it is to make correct predictions under the same level of pattern frequency. We checked the wrong predictions of this type (i.e., due to the required edit sequence being unusually long) in our case studies. We discovered that although there were exactly the same edit patterns in the training set, the model still made incorrect predictions in applying the pattern in the input clean code. The reason was intuitively because the successful injection requires the prediction on every edit operation to be correct, thus the overall probability of making the correct injection is the product of probabilities for making all those single-edit predictions correctly. Then, the more such single-edit predictions (as a result of a longer edit sequence) required, the lower that overall probability.

Vocabulary size. In additional case studies beyond the presented empirical evaluation, we experimented with the same dataset but of an enlarged vocabulary size. We then observed a noticeable decrease in the model performance. This can be explained by the fact that the increase in the vocabulary size enlarged the search space of the model during predictions of edits, leading to an increase in the output dimension of the logits/probabilities of the decoder. Consequently, the overall model complexity increased as well. Thus, how diverse the code tokens in the dataset underlying the model training and inference are plays a key role in the model's performance at the respective phases.

6.2 Technical Limitations

Despite the promising preliminary results obtained, our current technical design suffers a few limitations.

Incompleteness of design. When predicting the edit action a_t at time step t, we use the tree history $g_{1:t}$ and track it using LSTM. This increases the model complexity, and also makes the current action depend on historical trees/edits. On the one hand, this design may not be quite intuitive because as supervised by the "shortest-distance tree edits" in the training set, the model should ideally try to predict the shortest edit sequence given any intermediate tree g_t; that is, it should just need the current version of the tree being edited to determine how to make the next edit. On the other hand, with the current design, the quality of the predicted sequence might be more likely to rely on the first few edits. Thus, if there are multiple initial edits with comparable probabilities of getting predicted, the quality of the resulting sequence would suffer.

Scalability issue. When the training dataset contains large code snippets and each pair $\langle C_-, C_+\rangle$ has a large difference between the clean and vulnerable versions, the dynamic programming algorithm used to compute the ground-truth edits could be overloaded hence incur large time and space costs. The sample complexity could also be very high due to the incompleteness of design as discussed above. Moreover, when the vocabulary is large, the sample complexity is increased significantly since there is no name-independence [2] considered in the model design (i.e., the same token name appearing at different nodes of an AST is not always treated the same). The increased sample complexity would then lead to greater scalability challenges.

Lack of diversity. The current design has very limited capabilities of making diverse predictions (which are needed to produce more than one vulnerable sample for a given input clean sample). Thus, it is not sufficient for the diversity requirement of the general vulnerable sample generation problem. More importantly, the technique works by assuming the availability of C_+—the ground-truth target program, or at least a sample in the training set that is sufficiently similar to the input clean code (then the vulnerable version of that clean code in the training set can be used as the ground-truth C_+). First, this true target program is not known in advance in most cases. Second, the sufficiently similar pair may not be present in the training set either. Thus, the design assumptions limit the practical adoption of the technique, which needs to be lifted up or at least further relaxed in future work.

7 Conclusion

In response to recognizing the fatal consequences of software vulnerabilities to the cyber world, a large and still-growing number of defense techniques against particular types of vulnerabilities are devised. Yet there is a critical lack of scientific assessments of these techniques, which impedes systematical understandings of their strengths and limitations hence the fundamental advancement in software security assurance. A critical underlying cause is the lack of large-scale vulnerability datasets. To fill this gap, multiple challenges must be addressed, including (1) the insufficiency of empirical examples of research on automatically generating valid vulnerable programs, (2) the technical difficulty of producing vulnerable programs in terms of covered types of code vulnerabilities, and (3) the challenges to machine learning and data mining in inferring complex vulnerability semantics and vulnerability-inducing code-change patterns and using these as inputs to generate diverse code samples, especially with a limited number of training samples.

This chapter described, as the first step of large-scale realistic vulnerability data generation, an automated, learning-based approach to generating vulnerable samples from given clean code via vulnerability injection. Our approach learns a variety of code-change patterns that abstract/generalize incremental code changes that transform a program from its clean version to one of its possible vulnerable versions. It learns these patterns via a gated graph neural network based encoder taking the abstract syntax tree (AST) representation of programs as the model input, followed by an LSTM-based decoder to learn the sequence of code edits on the AST that meets the required program transformation. At testing time, these learned patterns are then applied to an arbitrary input clean code to generate vulnerable code.

Our preliminary evaluation has demonstrated the promise of learning-based vulnerability injection for vulnerable code generation: on a widely used dataset, the described approach was able to correctly inject vulnerabilities in 76% of the cases. Our efficiency results further suggest reasonable practicality of this approach in terms of model training time and peak memory cost. Meanwhile, we also extensively discussed how various aspects of the training dataset characteristics affect the model performance, as well as the technical limitations of our current design.

Acknowledgements The work described in this chapter was supported by an ARO award W911NF-21-1-0027. During our development of the described approach for vulnerability injection, we received very valuable comments and guidance from the authors of the Graph2Edit technique, for which we are grateful.

References

1. Allamanis, M., Brockschmidt, M., Khademi, M.: Learning to represent programs with graphs. In: International Conference on Learning Representations (2018)
2. Dinella, E., Dai, H., Li, Z., Naik, M., Song, L., Wang, K.: Hoppity: Learning graph transformations to detect and fix bugs in programs. In: International Conference on Learning Representations (ICLR) (2020)
3. Fu, X., Cai, H.: A dynamic taint analyzer for distributed systems. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1115–1119 (2019)
4. Fu, X., Cai, H.: Flowdist: Multi-staged refinement-based dynamic information flow analysis for distributed software systems. In: 30th {USENIX} Security Symposium ({USENIX} Security 21), pp. 2093–2110 (2021)
5. Ghaffarian, S.M., Shahriari, H.R.: Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. ACM Computing Surveys (CSUR) **50**(4), 56 (2017)
6. Harer, J.A., Ozdemir, O., Lazovich, T., Reale, C.P., Russell, R.L., Kim, L.Y., Chin, P.: Learning to repair software vulnerabilities with generative adversarial networks. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems, pp. 7944–7954 (2018)
7. Jiang, N., Lutellier, T., Tan, L.: Cure: Code-aware neural machine translation for automatic program repair. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 1161–1173. IEEE (2021)
8. Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R.: Gated graph sequence neural networks (2017)
9. Nong, Y., Cai, H.: A preliminary study on open-source memory vulnerability detectors. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 557–561. IEEE (2020)
10. Nong, Y., Cai, H., Ye, P., Li, L., Chen, F.: Evaluating and comparing memory error vulnerability detectors. Information and Software Technology **137**, 106614 (2021)
11. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al.: Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems **32**, 8026–8037 (2019)
12. Yamaguchi, F., Lindner, F., Rieck, K.: Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning. In: Proceedings of the 5th USENIX conference on Offensive technologies, pp. 13–13. USENIX Association (2011)
13. Yao, Z., Xu, F.F., Yin, P., Sun, H., Neubig, G.: Learning structural edits via incremental tree transformations. In: The Ninth International Conference on Learning Representations 2021 (ICLR'21) (2021)
14. Yasunaga, M., Liang, P.: Graph-based, self-supervised program repair from diagnostic feedback. In: International Conference on Machine Learning, pp. 10799–10808. PMLR (2020)
15. the Zephyr team: `http://asdl.sourceforge.net/` (2013)