Generating Realistic Vulnerabilities via Neural Code Editing: An Empirical Study

Yu Nong
Washington State University
Pullman, WA, USA
yu.nong@wsu.edu

Yuzhe Ou
The University of Texas at Dallas
Richardson, TX, USA
yuzhe.ou@utdallas.edu

Michael Pradel
University of Stuttgart
Stuttgart, Germany
michael@binaervarianz.de

Feng Chen
The University of Texas at Dallas
Richardson, TX, USA
feng.chen@utdallas.edu

Haipeng Cai∗
Washington State University
Pullman, WA, USA
haipeng.cai@wsu.edu

ABSTRACT
The availability of large-scale, realistic vulnerability datasets is essential both for benchmarking existing techniques and for developing effective new data-driven approaches for software security. Yet such datasets are critically lacking. A promising solution is to generate such datasets by injecting vulnerabilities into real-world programs, which are richly available. Thus, in this paper, we explore the feasibility of vulnerability injection through neural code editing. With a synthetic dataset and a real-world one, we investigate the potential and gaps of three state-of-the-art neural code editors for vulnerability injection. We find that the studied editors have critical limitations on the real-world dataset, where the best accuracy is only 10.03%, versus 79.40% on the synthetic dataset. While the graph-based editors are more effective (successfully injecting vulnerabilities in up to 34.93% of real-world testing samples) than the sequence-based one (0 success), they still suffer from complex code structures and fall short for long edits due to their insufficient designs of the preprocessing and deep learning (DL) models. We reveal the promise of neural code editing for generating realistic vulnerable samples, as they help boost the effectiveness of DL-based vulnerability detectors by up to 49.51% in terms of F1 score. We also provide insights into the gaps in current editors (e.g., they are good at deleting but not at replacing code) and actionable suggestions for addressing them (e.g., designing effective editing primitives).

CCS CONCEPTS
• Security and privacy → Software security engineering.

KEYWORDS
datasets, data generation, data augmentation, deep learning, software vulnerability, vulnerability detection, benchmarking

∗Haipeng Cai is the corresponding author.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9413-0/22/11.
https://doi.org/10.1145/3540250.3549128

1 INTRODUCTION
Software vulnerabilities constitute a major source of cybersecurity threats that can be exploited by security attacks leading to information leakage, software crashing, and data tampering, among other consequences [13]. In response, a variety of technical approaches defending against software vulnerabilities (e.g., [19, 31, 41]) have been proposed, of which the most momentous are those based on deep learning (DL) [20, 25]. Indeed, DL-based software analysis in general [47], and software vulnerability detection [9, 35, 36, 65] and repair [22] in particular, have achieved great successes, reporting accuracies that often surpass traditional approaches.

However, these promising-looking accuracy results are often obtained on synthetic, rather than real-world, programs, due to a critical lack of large-scale and realistic datasets. In particular, there are not enough vulnerable code samples for which we know the vulnerability ground truth. This leads to two immediate barriers against advancing software assurance against vulnerabilities:

• Benchmarking: fair and real-world evaluation of existing techniques. The lack of a realistic, sizable dataset leads to the inability to benchmark existing techniques fairly in a realistic setting (i.e., working effectively on real-world software—the ultimate goal of the techniques). Current techniques are either evaluated on synthetic benchmarks only, or their accuracy becomes much lower [9, 65]. Meanwhile, existing comparative studies of the techniques are generally limited to technical discussion and qualitative assessment [2, 28, 52] and/or incomplete comparisons (e.g., just comparing the numbers of vulnerabilities found rather than precision and recall) [3–5, 46].

• Model training: development of new and more effective DL-based techniques. DL-based techniques for vulnerability analysis have shown great promise. Yet their accuracy is commonly not up to the mark in real-world application settings. The main reason is that they are not trained in such settings due to the lack of large-scale, realistic datasets. Prior work has clearly shown
that the lack of sufficient training data is a major barrier to high accuracy of DL-based software vulnerability detection \cite{15,38} and localization \cite{34}.

Some software vulnerability datasets are available. SARD \cite{7} and SATE IV \cite{43} provides a large number of (60,000+) vulnerable code samples and their fixed versions. However, these samples are synthetic and not representative of real-world vulnerability analysis situations. CVE/NVD \cite{8} is a high-quality database of vulnerabilities in real-world projects, but the corresponding buggy and fixed code is not easily collectable. Several studies \cite{6,17} tried to address this, but the numbers of collected code samples are too small (<5,000) for training effective DL models. Therefore, others \cite{18,29,64} attempted to garner vulnerability data in the wild. Yet the effort is to retrieve historical vulnerable versions of given projects, hence the outcome is limited to existing data, which is what we lack. Zhang et al. \cite{63} aimed to generate null-pointer-dereference vulnerability code samples, but it is unclear whether the data is realistic, and the samples only represent one vulnerability type.

To address this critical gap, an intuitive solution is to automatically build a large-scale, realistic vulnerability dataset. In particular, a promising direction is to generate vulnerable programs by injecting vulnerabilities into real-world (presumably non-vulnerable) programs, which are widely available. Given the success of neural code editors \cite{11,16,59}, i.e., neural models trained to transform code, on other tasks, applying them to vulnerability injection seems promising. However, it currently remains unclear whether neural code editors can effectively inject vulnerabilities and whether doing so would provide useful training data to DL-based vulnerability analyses.

This paper presents an empirical study aimed at addressing these questions. Specifically, we study the ability of state-of-the-art neural code editing models to produce realistic vulnerabilities, and whether adding such DL-generated vulnerabilities to a training dataset improves the effectiveness of learning-based vulnerability detectors. With a commonly used synthetic dataset and a real-world dataset, we investigate the potential and gaps of three state-of-the-art neural code editors of different kinds (sequence-based and graph-based) for vulnerability injection. We use both synthetic and real-world datasets because we want to compare the difficulties of generating synthetic versus realistic vulnerabilities. We start by investigating the technical strengths and weaknesses of these editors using the synthetic dataset with various modifications, from data characteristics and model architecture/algorithms perspectives. Then, we evaluate the effectiveness of these editors in generating realistic vulnerable samples, while assessing the realism of the samples via a user study. Finally, to validate the practical potential of the injection approach, we use the DL-generated realistic samples to augment two existing real-world datasets used for training two state-of-the-art vulnerability detectors and evaluate the gains in their accuracy at detecting vulnerabilities in other real-world programs.

Among other findings, our study reveals that:

- While much more useful, realistic vulnerable samples are also much harder to generate than synthetic ones (best accuracy of 10.03% versus 79.40% with any of the editors), due to challenges like larger vocabulary size (#unique code tokens), greater program length, and more complex structures of realistic programs.

- The graph-based editing techniques (which predict edits incrementally) are more effective (achieving up to 34.93% success rate) than the sequence-based one (which predicts target code itself, and with no success) in generating realistic vulnerabilities.

- The graph-based techniques are much better at deleting code than adding/replacing code. In over 55% of the success cases, they only delete code line(s), while only 41.19% of the ground-truth injections delete line(s) only. In less than 45% of the success cases, they replace/add code, which is needed for successfully injecting the ground-truth vulnerabilities in most (58.81%) cases.

- By adding our DL-generated samples to their training sets, the improvements of two state-of-the-art DL-based vulnerability detectors over their baseline are significant (by up to 49.51% in F1), much higher than the ones (up to 10.40%) by using the same number of synthetic samples (as a lower bound of such improvements), but still lower than the improvements (up to 68.76%) by using real-world samples (as an upper bound).

Based on these findings, we provide actionable suggestions for improving neural code editing techniques in general and for generating realistic vulnerabilities in particular. Among other recommendations, we suggest to: (1) reduce the edit search space by predicting incremental edits rather than the target code itself; (2) use graphs rather than sequences to represent programs; (3) design code editing primitives based on task characteristics (e.g., the granularity of edit prediction should match that of a task like code editing); and (4) improve and develop techniques to ensure the quality (e.g., realism, diversity, and low noise) of the generated data.

Open science. Source code and datasets are all available in our artifact and have been made publicly accessible.

## 2 METHODOLOGY

This section elaborates the design of our study, including the research questions, neural code editors chosen, and datasets used.

### 2.1 Research Questions

We seek to answer three questions, as outlined and justified below.

**RQ1. What are the strengths and weaknesses of the editors?**

We start with a detailed technical review of the editors (through the related papers) and empirical experiments of their effectiveness on synthetic code to understand their strengths and weaknesses. The rationale is that this understanding will help assess the potential and gaps in them for realistic vulnerability injection. The reasons to use the synthetic dataset are two-fold. First, given the great complexity and diversity of real-world code, results on synthetic code can inform about an upper bound of the effectiveness of these editors. Second, the synthetic nature enables us to experiment with various modifications (e.g., automated code refactoring, variable renaming) of the dataset to assess the generalizability of editors, and hence, gain deeper insights into their potential and limitations. Doing such modifications on real-world programs could make them unrealistic—which is also why we only use the synthetic dataset for RQ1.

**RQ2. Can the editors generate realistic vulnerable code?**

With the understanding obtained from RQ1, we then assess how well the editors generate realistic vulnerable code and what conditions...
make them (not) work. This RQ is readily justified by the main goal of our entire study—the results and insights from answering it will immediately reveal how far we are in the direction of using neural code editing for generating realistic vulnerabilities.

**RQ3. Does the generated vulnerable code help improve the DL models for downstream vulnerability analysis tasks?** One of the key premises of pursuing the direction of DL-based vulnerability generation is that such generated datasets are practically useful in downstream application tasks for software assurance against real-world vulnerabilities. We target DL-based vulnerability detection as the task in this paper given its critical role in defenses and vital importance of quality training data in building such detectors.

### 2.2 Neural Code Editors

We perform a literature review on neural code editing, and then select editors for our study per the criterion below.

1. **Availability**: the editor’s source code must be publicly available.
2. **Standalone**: the editor requires no inputs beyond the code under vulnerability injection (e.g., no commit logs or error messages from other tools), so that the injection can be more applicable.
3. **Coverage**: per our literature review, mainstream editors are either graph- [1, 16, 59, 60] or sequence-based [6, 10, 11, 22, 24]; we thus intend to cover the state-of-the-art in both categories.

As a result, we chose Graph2Edit [59] and Hoppity [16], two latest graph-based editors, and SequenceR [11], a state-of-the-art sequence-based editor, that meet all the above criteria. Next, for each editor, we briefly summarize the design choices relevant to our study and how we set it up. For more details, we refer readers to the respective original papers.

**SequenceR** [11] is an NMT-based [27] sequence-to-sequence code editor for program repair. It works by tokenizing the given buggy program into a sequence to tokens, and then translating the sequence to a fixed code sequence. Users can specify the buggy code fragment by adding special tokens “<START_BUG>” and “<END_BUG>” before and after it, respectively. For better accuracy, it uses the **copy mechanism** to address the unlimited vocabulary problem [50]. In our study, since the samples do not have specified code fragments to inject vulnerabilities, we simply surround the entire program with the two special tokens.

**Graph2Edit** [59] is a code editor taking the abstract syntax tree (AST) of a given program and producing its transformed version by predicting and applying a sequence of AST edits. Each editing operation (edit action) consists of one or more of three elements: the operator type, node to be edited, or the type/value of the node. Three types of operators are considered: adding a node, deleting a node, and copying a subtree. The editor first converts the AST to a graph by adding bidirectional edges between adjacent sibling nodes as well as parent and child nodes. Then, it learns the graph and node embeddings using a gated graph neural network (GGNN) [32], and predicts a sequence of edit actions based on the embeddings using a long short-term memory (LSTM) network [59]. It also uses an edit encoder to encode the edits between the input and target code to assist with the edit prediction. To improve accuracy and efficiency, for each pair used to train the model, Graph2Edit computes the shortest AST edit sequence as ground-truth using a dynamic programming (DP) algorithm. The model is trained to maximize the probability of predicting the shortest edit sequence.

For vulnerability injection, the target code as part of the inputs to the edit encoder is not supposed to be known/given beforehand. Thus, for our study we disabled the edit encoder, following guidance by the Graph2Edit authors, by feeding nothing to the encoder and making it constantly output a zero vector. To adapt this editor originally designed for C# to our study, we use Joern [58], a robust AST parser for C language, to obtain the ASTs of our input samples.

**Hoppity** [16] is code editor aiming to fix bugs in JavaScript programs. Compared to Graph2Edit, its input is also the AST of a given program and it transforms the program also via a sequence of edits. Each edit action includes three elements: node location, node value, or the node type, and four types of edit actions are considered: adding a node, deleting a node, replacing node type, and replacing node value. Hoppity also augments the input AST to a graph, but in a different way: the leaf nodes are connected by edges called **suc links**, and the so-called **value links** are used to connect the AST nodes to associated values stored in a local value table. It uses the graph isomorphism network (GIN) [57] to learn the graph and node embeddings, and LSTM to predict edit actions like Graph2Edit does. Yet Hoppity does not use the DP algorithm, but a NodeJS plugin ShiftParser [51], to obtain the ground-truth edit sequences. For our study, we also use Joern to generate the ASTs.

To adapt these editors for vulnerability injection, we train them on pairs of vulnerable samples and corresponding fixed (normal) versions. At testing time, we feed the trained model with normal samples as inputs and expect to obtain the associated vulnerable versions as the outputs. For an input sample X, we consider the output Y **accurate** if Y’s AST exactly matches that of the vulnerable sample paired with X; thus, accuracy = #accurate outputs/#inputs.

### 2.3 Datasets

We use two datasets, one synthetic and the other real-world, to assess the three neural code editors extensively.

**Synthetic dataset.** We use SARD [7], which includes a large and commonly used set of vulnerable samples and the respective fixed (normal) versions, to build our synthetic dataset. For our study, we only select the samples written in C. As the editors used only support code editing for functions, we remove the pairs of samples where the code changes are outside functions (e.g., macros, global variables). Finally, we obtain 15,943 pairs of code samples covering 90 types of vulnerabilities, where the average number of lines of code is 31.59 and the average number of changed lines is 4.91.

**Real-world dataset.** We curate a real-world dataset based on BigVul [17] and PatchDB [56]¹, which includes 3,754 and 12,073 patches, respectively, all from real-world projects. Each patch has a pair of vulnerable and the respective fixed (normal) samples. We only select C samples and remove those where the code changes are outside functions. Since many real-world patches are very complex and make the code editors cost too much time and memory to process them, we remove the patches where the vulnerability injection requires more than 100 AST edits based on Graph2Edit’s

---

¹The BigVul used as a basis of our real-world dataset is the collection of C/C++ samples from the CVE/NVD database [30]. As BigVul alone is relatively small, we additionally considered another source (PatchDB) to form our real-world dataset.
Table 1: RQ1: Accuracy on the original synthetic dataset, and with different treatments for testing samples.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Original</th>
<th>Large Vocab</th>
<th>Complex Code Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hoppity</td>
<td>59.22%</td>
<td>48.65% (17.85%)</td>
<td>49.45% (16.56%)</td>
</tr>
<tr>
<td>Graph2Edit</td>
<td>79.40%</td>
<td>13.13% (83.46%)</td>
<td>40.71% (48.73%)</td>
</tr>
<tr>
<td>SequenceR</td>
<td>72.28%</td>
<td>31.59% (56.28%)</td>
<td>2.42% (96.65%)</td>
</tr>
</tbody>
</table>

DP algorithm [59]. We finally obtain 7,789 patches covering more than 340 open-source projects, where the average number of lines of code is 72.84, and the average number of changed lines is 4.92. To avoid taking too much time for training the editors, only the functions related to the vulnerability are retained in each patch.

To illustrate “synthetic” versus “real-world” code, Figures 3 and 6 show some examples for these two kinds, respectively. As seen, the real-world sample is much more complex than the synthetic.

Our experiments were performed on a server which had an AMD Ryzen Threadripper 3970X (3.7GHz) CPU with 32 Cores, an Nvidia GeForce RTX 3090 GPU, and 256GB memory.

3  RQ1: STRENGTHS AND WEAKNESSES

We start with the effectiveness of the editors on the original synthetic dataset, then examine their generalizability against modifications of the dataset, and finally the performance impact of key dataset factors. For all the experiments for RQ1, we split the 15,943 pairs of samples into 80%:10%:10% for training, validation, and testing.

3.1 Results on the Original Synthetic Dataset

Table 1 (first two columns) shows the accuracy of the three editors. Graph2Edit achieves the highest accuracy (79.40%), followed by SequenceR (72.28%) and then Hoppity (59.22%).

Hoppity is much less accurate than Graph2Edit despite the fact that they have similar designs. One of the reasons is its suboptimal preprocessing (with ShiftParser) that generates redundant and inefficient ground-truth edit sequences. As Figure 1 (Edit Sequences 1) shows, Graph2Edit generates the optimized one-action edit sequence with its DP algorithm, while Hoppity generates a redundant one (15 actions). Another factor that contributes to the differences here is the design of the edit operators — Graph2Edit has a special operator copying a subtree while Hoppity does not, as illustrated in Figure 1 (Edit Sequences 2). With this operator, Graph2Edit finishes the editing in two actions, while Hoppity needs 44. Since Graph2Edit and Hoppity predict the edit sequence in an iterative manner, which is sensitive to the number of editing steps, the redundant ground-truth edit sequences makes it harder for Hoppity to predict the overall edits correctly, significantly limiting its effectiveness.

The higher accuracy of Graph2Edit over SequenceR indicates that graph-based approaches are more effective than the sequence-based ones for vulnerability injection. Different from natural-language texts, computer programs are highly structured. The syntactic and semantic information of programs is hard to model via sequences, but can be more precisely represented by graphs, which helps with vulnerability injection.

3.2 Generalizability of the Editing Techniques

To examine the generalizability of these editors to the diverse and complex realistic samples, we perform two extended experiments.

The first experiment, called Large Vocab, evaluates the impact of vocabulary size (#unique tokens) on the effectiveness of the editors. As in [62], for each pair of samples in the testing set, we randomly replace the identifiers (function/variable names) with unrelated names (e.g., \texttt{int bufsize=0} changed to \texttt{int h3d2k=0}), but also consistently so that the program semantics/functionalities do not change. This increases the vocabulary size from 1,401 to 5,422. Then, we test the trained models in Section 3.1 on the modified testing samples.

Table 1 (3rd column) shows the results. All the techniques suffer from an accuracy decrease in this experiment, because a larger vocabulary size means a larger search space for the tokens. Graph2Edit has the largest accuracy decrease (83.46%), because it does not consider identifier independence\(^2\), which separates the identifier names from the prediction explicitly. In contrast, Hoppity is much more generalizable, with only a 17.85% decrease, thanks to its enhanced AST representation, where the local value table and the value nodes in the table enable identifier independence. Relative to Graph2Edit, SequenceR is more generalizable due to its copy mechanism [21], which helps achieve identifier independence and enables the model to copy unseen tokens when predicting edits.

The second experiment, called Complex Code Structure, evaluates how more complex code structures impact the effectiveness of the editors. For each pair of samples in the testing set, we refactor the code samples and add unrelated code. For refactoring, we (1) reverse the condition in the if-statements and then exchange the bodies of the if-statement and the else-statement, as illustrated in Figure 2 (Strategy 1 and 2) and (2) convert the for-loops into while-loops, as Figure 2 (Strategy 3) shows. For adding unrelated code, we randomly add variable declarations and assignments to them, as Figure 3 shows. These changes are also ensured to be semantics-preserving. Then, we test the trained models on the modified testing samples.

Table 1 (4th column) shows the results. All the techniques suffer from accuracy drops with larger vocabulary sizes, which could be mitigated by ensuring identifier independence.

\(^2\)The ability of a learning-based code model to learn the code semantics without being impacted by variations of identifier names.
of the two child subtrees of the if-statement. Inserting unrelated statements only adds a subtree without affecting other parts of the AST. The GIN also helps here as it outputs the same embeddings which determine the edit predictions for two isomorphic graphs while a refactored AST may be isomorphic to its original AST.

With a 40.71% accuracy, Graph2Edit is less generalizable. The reason is that, while also using ASTs, the GGNN used by Graph2Edit cannot handle code refactoring changes as the GIN does.

Sequencer almost fails with a 2.42% accuracy as the sequence-based model has no mechanism to deal with code structure changes.

All the techniques have lower accuracy with more complicated code structures. Sequence-based models are most sensitive to this problem, whereas AST and graph-based models help ensure generalizability.

### 3.3 Impact of Key Dataset Factors on Accuracy

We consider three dataset factors key to our study: (1) the edit length of a testing pair, measured as #edit actions, (2) the program length of a testing pair, measured as #tokens in the ground-truth vulnerable sample, and (3) the pattern frequency of a testing pair, measured as #tokens in the ground-truth vulnerable sample, and (3) the pattern frequency of a testing pair, measured as #tokens in the ground-truth vulnerable sample.

Table 2 shows the impact of edit length on the prediction accuracy. The accuracy decreases when the edit lengths increase, for all of the techniques. For Hoppity and Graph2Edit, this is because they predict edit actions iteratively; thus, the larger the edit length, the larger the probability that the overall prediction is incorrect. For Sequencer, while the edit lengths may not directly impact the accuracy, the positive correlation between edit length and program length, and the negative correlation between edit length and pattern frequency, still (indirectly) lead to accuracy decreases.

Table 3 shows the impact of pattern frequency on the prediction accuracy. For all the techniques, the accuracy increases when the pattern frequency increases. This can be explained by the nature of the gradient descent (GD) algorithm [49] used for model training: The more often one pattern is observed in the training set, the more likely the models predict correctly on it. The 2nd column shows an extreme case: The techniques cannot predict correctly on those testing pairs with unseen patterns.

Table 4 shows the impact of program length on the prediction accuracy. Generally, the accuracy on longer programs is lower with these techniques. For Hoppity and Graph2Edit, the AST size grows as the number of tokens increases, making the message passing in the GNNs harder. For Sequencer, longer programs mean more tokens to predict, making it harder to predict all tokens correctly.

The accuracy of all the techniques generally decreases significantly when the edit length or program length increases, or when the pattern frequency decreases. This property of the models limits their effectiveness at injecting complex vulnerabilities.
To answer the question whether the editors can generate realistic vulnerable code, we split the 7,789 pairs of samples in the real-world dataset into 60%:10%:30% (4,678:778:2,333) for training, validation, and testing. The reason for keeping 30% of the data for testing is that we want to use more samples in RQ3.

4.1 Results on the Real-World Dataset
Table 5 (2nd column) shows the exact match accuracy on our real-world dataset. The results are much worse than those on the synthetic dataset. To help understand the contrast, Figure 5 shows the much greater complexity of the real-world dataset than the synthetic one in terms of larger vocabulary size, greater program length, and lower pattern frequency. The edit length of the real-world dataset is smaller because we filter the dataset (Section 2.3) by removing samples with long edit lengths (>100 edit steps), which each would need unaffordable time (~1hr) and/or all the GPU memory (24GB) for the editors to process. For RQ1, we have discussed the impact of these factors with the synthetic dataset. With the real-world dataset, these impact are even larger, further indicating the gaps with existing editors from realistic vulnerability data generation.

Graph2Edit achieves the best accuracy, largely due to its DP algorithm and copy a subtree operator, which reduce the edit lengths. In contrast, Hoppity has lower accuracy because of its suboptimal preprocessing that generates redundant ground-truth edit sequences. Yet, its use of ASTs, identifier independence, and the GIN still help it achieve a 1.41% accuracy. SequenceR fails entirely on all the real-world samples. Our manual inspection reveals that it cannot generate any (syntactically) valid programs. The main plausible reasons are that (1) the sequence-based model is very sensitive to program lengths, whereas the real-world programs are usually very long, as Figure 5 shows; (2) the model has no explicit mechanism to reason about code syntax and semantics; and (3) the model is sensitive to complex code structures, which the real-world samples usually have [9].

The failure of SequenceR indicates the significant limitation of sequence-based editors for vulnerability injection. While such editors are successful at bug fixing, they highly depend on mature techniques for bug localization, which allows them to reduce the sequence to be generated and take advantage of the sequence-based models on natural languages [6, 11]. However, there is no technique to analyze the normal samples and localize the code fragments to inject vulnerabilities yet. Thus, current sequence-based editors leave room for improvement for generating vulnerabilities.

All the techniques have underwhelming accuracy on the real-world dataset because it is much more complex in terms of larger vocabulary size, greater program length, and lower pattern frequency, which significantly impact the accuracy, indicating their gaps for realistic vulnerability generation.

So far, we have measured accuracy with respect to the existing ground truth. However, there may be multiple ways to inject a vulnerability into a given code sample. To account for this situation, we manually check each DL-generated code sample that does not match its ground truth, indicating their potential for generating realistic vulnerability generation.
Table 6: How do the kinds of edits (add/remove/replace lines) done by the models and those in the ground truth differ?

<table>
<thead>
<tr>
<th>Tool</th>
<th>Edits % Add line(s) only</th>
<th>% Delete line(s) only</th>
<th>% Replace line(s) only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ground truth</td>
<td>All</td>
<td>10.9%</td>
<td>41.19%</td>
</tr>
<tr>
<td>Graph2Edit</td>
<td>All</td>
<td>0.13%</td>
<td>96.66%</td>
</tr>
<tr>
<td></td>
<td>Success, exactly-match</td>
<td>0.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td></td>
<td>Success, not exactly-match</td>
<td>0.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Hoppity</td>
<td>All</td>
<td>1.41%</td>
<td>14.88%</td>
</tr>
<tr>
<td></td>
<td>Success, exactly-match</td>
<td>3.63%</td>
<td>57.58%</td>
</tr>
<tr>
<td></td>
<td>Success, not exactly-match</td>
<td>4.47%</td>
<td>59.78%</td>
</tr>
</tbody>
</table>

4.2 Case Studies
Based on our manual inspection of the DL-generated samples, we notice some patterns in the generation process. Thus, we quantitatively investigate two questions about the behavior of the editors. Since SequenceR could not generate (syntactically) valid samples in our testing (Section 4.1), we perform these case studies only on Graph2Edit and Hoppity.

Q1: How do the edits (add/remove/replace lines) done by the models and those expected in the ground truths differ?
Since the editors inject vulnerabilities via code editing, we want to know what kinds of edits they typically perform and which of these edits successfully inject vulnerabilities. To be consistent across Graph2Edit and Hoppity, we use the diff tool [39] to get the lines added/deleted between the vulnerable samples and the normal ones. We then examine how often the models add line(s) only, delete line(s) only, or do both (i.e., replace line(s)), and compare these numbers to the ground truth edits.

Table 6 shows the results. Of the ground-truth edits, more than half (57.74%) are replacing line(s) and 41.19% are just deleting line(s). Merely 1.07% of the edits are adding line(s) only.

In comparison, Graph2Edit only deletes line(s) for almost all the samples (99.66%), which also covers the entire set of success cases. This indicates that it tends to inject vulnerabilities by deleting code and is good at it. Since more than 40% of the ground-truth edits are also deleting line(s) only, Graph2Edit takes this advantage and performs the best in both the exactly-match accuracy and success rate. However, this also indicates that Graph2Edit may not be very good at replacing code. The reason is that Graph2Edit does not have the node replacing operators as Hoppity does, making the edit sequence for code replacing longer hence reducing the effectiveness.

In contrast, Hoppity replaces lines in most of the edits (83.71%). Even in the success cases, more than 30% of the edits are replacing line(s) (39.39% for exactly-match cases and 35.75% for success but not exactly-match cases). Compared with Graph2Edit, Hoppity takes advantage of its replacing node value and replacing node type operators, which reduce the edit sequence length. By manually inspecting the success cases of Hoppity, we notice that most of them are replacing one or several node values (e.g., replacing "=" to ">=").

While Hoppity is better at replacing line(s) than Graph2Edit, the proportions of deleting line(s) only edits in the success cases (>55%) are still much higher than the ones in the ground-truth edits (41.19%) and all of Hoppity’s edits (14.88%). This indicates that it is still easier to successfully inject vulnerabilities by deleting code, even for Hoppity. In fact, the deleting a node operators in both Graph2Edit and Hoppity are simpler than other operators. Compared with

Q2: Do the neural code editors tend to generate generic vulnerabilities or domain-specific ones?
For high data quality, it is desirable that the DL-generated samples are diverse but also have similar distributions to real-world samples. One way to examine these properties is to differentiate the vulnerabilities between generic and domain-specific. We refer to a generic vulnerability a bug that may cause security issues in any domain of software (e.g., buffer overflow, dangling pointer, integer overflow), and a domain-specific vulnerability a security bug only in some specific software domains (e.g., improper authentication, improper input validation, insufficiently protected credentials). To illustrate, Figure 6 shows an example of each kind.

All the techniques are better at deleting lines than adding or replacing code. A better design should improve at other types of operations while keeping that strength.

Figure 6: Generic versus domain-specific vulnerabilities.
identified and may have bias (e.g., we may miss more domain-specific vulnerabilities since they are harder to identify).

Table 7 shows the proportion of domain-specific vulnerabilities in the ground-truth samples and those in the exactly-match samples produced by Graph2Edit and Hoppity. We notice that both editors tend to generate fewer domain-specific samples with respect to the ground truth. The reason is that the code relevant to generic vulnerabilities is easier to identify by the models. We notice that the code relevant to generic vulnerabilities has many similarities. For example, many of them use common identifiers like buf, buflen, len, length or involve relational expressions, as Figure 6 shows. However, the domain-specific vulnerabilities are much more diverse. The identifiers used are rarely the same and the code of the vulnerabilities is diverse (e.g., code for improper authentication, improper input validation, and insufficient protected credentials may be very different). This indicates that the editors may be weaker at generating domain-specific vulnerabilities, compromising the overall quality of the DL-generated data.

In comparison, Hoppity generates a much smaller proportion of domain-specific vulnerabilities than Graph2Edit. The reason is that Hoppity tends to replace line(s) in most of the cases, usually by replacing tokens. Since generic vulnerabilities often involve common tokens and node types, it is much easier to inject such vulnerabilities by just replacing tokens and node types. In contrast, the code diversity of domain-specific vulnerabilities makes it much harder to inject them by simply replacing tokens and node types.

Both techniques generate smaller proportions of domain-specific vulnerabilities than generic ones. This contrast, compared to that in the ground truth, suggests a relative weakness of these two neural code editing techniques in generating domain-specific vulnerabilities.

4.3 User Study
To evaluate the realism of the success samples that do not exactly match ground truth, we conduct a user study for which we randomly select 20 of them and 20 vulnerable samples in the real-world dataset. As for our case studies, we used random sampling without replacement (assuming that each sample has the same probability of being selected) to obtain these samples. Given the high cost of such manual studies, we only sampled once. We then shuffle these 40 samples for each participant, who is asked to rate their confidence (out of four levels, where four is the highest) that the vulnerability in each sample is realistic (i.e., made by a developer rather than an editor) in each sample. The participants are told what and where each vulnerability is in the code.

Six participants completed the study, who have at least 3 (mostly 10–15) years of experience with software engineering and security. Figure 7 shows the average of their confidence levels for each sample. We use Wilcoxon signed-rank tests [55] to compute the statistical significance (p values), and Cliff’s Delta [14] as effect size, of realism differences between the real-world and DL-generated samples. Our results (p = 0.624, effect size = 0.042) indicate that the DL-generated samples have no statistically significant or large difference from real-world samples in terms of realism.

5 RQ3: USEFULNESS OF GENERATED DATA
In this RQ, we investigate whether our DL-generated vulnerable samples help improve DL-based vulnerability detectors that predict whether a given sample is vulnerable. We choose two state-of-the-art detectors, Devign [65] and ReVeal [9], as they are considered the most effective such detectors for C so far [9, 33]. Their original datasets are summarized in Table 8. Since the ultimate goal of a vulnerability detector is being able to detect unseen vulnerabilities in real-world software, we apply independent testing for our experiments, i.e., the testing and training samples are from different datasets. To test the detectors more comprehensively, we further use a third-party dataset Xen, which is a subset of the dataset introduced in [37] as an additional testing set, also shown in Table 8.

Using these datasets, we consider three experiment settings:

1. Reproduction: We use the same experiment setting used in [9]: the training set is Devign and the testing set is ReVeal.
2. Partial replication: We keep the training set Devign used in [9] and only change the testing set to Xen.
3. Full replication: We change both the training set and the testing set by using ReVeal for training and Devign for testing.

We use these three settings against the two detectors as baselines. Then, we improve their training set by adding the 2,333 × 34.93% = 815 success samples from Graph2Edit (as it is the most effective editor), and test whether the re-trained models perform better. Note that the testing set is kept the same in this process. Since these added samples are all labeled as vulnerable, to avoid the impacts of data balance changes, we add proportional numbers of real-world normal samples from [37] (others than those already included in the Xen dataset above) such that the balance does not change.

Table 9 column Baseline shows the effectiveness of the two detectors using the original training sets and column Improved shows the relative improvements after adding the samples to the training sets. For example, in reproduction, Devign’s F1 improves from 16.83% to 19.31%, i.e., by 14.74%. In any setting, all the metrics (precision, recall, F1) improve significantly after adding the DL-generated samples, except for a 3.67% decrease in Devign’s precision in full.
Table 9: RQ3: Does the DL-generated samples help improve the DL-based vulnerability detectors?

<table>
<thead>
<tr>
<th>Tool</th>
<th>Setting</th>
<th>Metric</th>
<th>Baseline</th>
<th>Improved Synthetic</th>
<th>Ground Truth</th>
<th>All Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reproduction</td>
<td>Precision</td>
<td>10.75%</td>
<td>9.67%</td>
<td>9.95%</td>
<td>8.56%</td>
<td>3.35%</td>
</tr>
<tr>
<td>Training: Devign</td>
<td>Recall</td>
<td>38.78%</td>
<td>37.26%</td>
<td>11.76%</td>
<td>56.88%</td>
<td>57.86%</td>
</tr>
<tr>
<td>Testing: ReVeal</td>
<td>F1</td>
<td>16.83%</td>
<td>14.74%</td>
<td>10.40%</td>
<td>16.34%</td>
<td>11.76%</td>
</tr>
<tr>
<td>Partial Replication</td>
<td>Precision</td>
<td>8.73%</td>
<td>16.04%</td>
<td>-10.31%</td>
<td>30.81%</td>
<td>5.04%</td>
</tr>
<tr>
<td>Training: Devign</td>
<td>Recall</td>
<td>37.48%</td>
<td>102.49%</td>
<td>-1.52%</td>
<td>50.24%</td>
<td>26.12%</td>
</tr>
<tr>
<td>Testing: Devign</td>
<td>F1</td>
<td>14.16%</td>
<td>26.20%</td>
<td>-8.76%</td>
<td>54.11%</td>
<td>8.47%</td>
</tr>
<tr>
<td>Full Replication: Precision</td>
<td>55.56%</td>
<td>3.67%</td>
<td>0.92%</td>
<td>1.64%</td>
<td>7.81%</td>
<td>3.14%</td>
</tr>
<tr>
<td>Full Replication: Recall</td>
<td>2.76%</td>
<td>38.04%</td>
<td>8.70%</td>
<td>74.28%</td>
<td>5.43%</td>
<td>5.43%</td>
</tr>
<tr>
<td>Full Replication: F1</td>
<td>5.25%</td>
<td>35.43%</td>
<td>8.57%</td>
<td>86.76%</td>
<td>5.52%</td>
<td>5.52%</td>
</tr>
</tbody>
</table>

We further discuss what properties of neural code editors makes them effective at generating realistic vulnerabilities.

6 DISCUSSION

6.1 Incremental Edit

A key design factor of DL models is the output space. We evaluate both sequence- and graph-based models and show that the latter can be more effective (Sections 3.1 and 4.1) and generalizable (Section 3.2). Besides the general merits of graph-based models, predicting target edits (as what Hoppity and Graph2Edit do) rather than target programs has several advantages: (1) modeling incremental edits better mimics the behavior of human in code editing—developers make code changes incrementally; (2) the search space of an edit (sequence) is much smaller than that of a target program and more decomposable to lower-level primitives; and (3) smaller search space also implies less data needed to train and activate the neural models.

Decomposing the search space into lower-level editing primitives also makes the resultant models more interpretable. When lower-level primitives are produced by the model, it is clear what the model is doing at each step and how it behaves to generate certain types of vulnerability (e.g., buffer overflows). We discuss further about the design of these editing primitives (Section 6.3).

Thus, we suggest DL-based vulnerability generation approaches learn predicting incremental edits rather than the changed code.

6.2 Data Representation

As we discuss in Sections 3.1 and 3.2, SequenceR uses text sequences while Graph2Edit and Hoppity use AST-based graphs to represent programs. Despite Graph2Edit having the best accuracy among the three editors on both datasets, the comparison between Hoppity and SequenceR seems also interesting. Although Hoppity has lower accuracy than SequenceR on the synthetic dataset (Table 1), it performed better on the realistic dataset (Table 5), demonstrating the advantage of a more structured representation when data complexity increases.

A high-level explanation is that ASTs have richer and easily accessible information (syntactic structure) than text sequences, which is important for vulnerable code generation. The GNN also helps, as it only takes a few steps of message propagation, and hence, avoids overfitting by restricting message passing to local structures. While the LSTM in SequenceR allows forgetting and dynamically adjusting its memory, the model can still learn noisy long-distance correlations between tokens.

Yet, ASTs do not contain easily accessible semantic information behind the syntactic structure. As a vulnerability is more related to semantic than syntactic information, representations that incorporate semantic information (e.g., control/data flow) may be better.
6.3 Code Editing Primitives

In Section 3.1, we find that one main weakness of Hoppity is its inability to predict a primitive operation for copying a subtree in the AST. This contributes to Hoppity’s limited accuracy because the common traits of code editing are not considered in the design of its edit primitives. When the code is represented as a tree like AST, a relatively small change to a code line translates to changes in a subtree \( t \) (which represents the code line) where most parts (a subtree \( tt \) of \( t \)) often remain unchanged. Thus, a model learning to predict individual (token-level) edits to build \( tt \) is apparently cumbersome and error-prone (the more edits to predict, the less likely to correctly predict them all). Accordingly, we suggest future models design should consider copying a subtree as a direct, lowest-level edit primitive when the code representation is a tree. At a higher level, the granularity of edit prediction should respect (i.e., be considered based on) the granularity of changes to the particular code representation rather than to the code itself (as a text sequence). More generally, this Hoppity weakness represents a mismatch between the design of the model/algorithm and the characteristics of its application domain (e.g., code editing). We suggest to design edit primitives based on such characteristics to avoid the mismatch.

Compared to Graph2Edit, another weakness of Hoppity is its ineffective preprocessing: instead of training the network against just one direct edit for deleting a code line, it feeds the network with an excessive number of edits that realize the deletion through cascading replacement of a line with the line below it (starting from the bottom of the AST all the way until the line to be deleted). Seemingly an implementation issue, this generally represents a major design flaw (in Hoppity)—it overly burdens the model with learning tasks of unnecessary complexity hence downgrades the model accuracy (Section 3.1). We suggest to shift such burdens of low-level (e.g., editing) operations to a deterministic process outside the network, hence minimize the decisions to be made by the NN model. Moreover, we suggest to let the model focus on learning the most essential, probabilistic steps (e.g., predicting which line to delete), while offloading deterministic steps (e.g., actually deleting the line) from the model itself. As one example of validating the merits of these suggestions, we implemented a dynamic-programming algorithm to reduce the average (ground-truth) edit length by 33%, which resulted in noticeably more accurate edit location prediction with Hoppity. As another example, we also realized the insight of learning the most essential probabilistic steps to separate edit localization and editing, by removing the irrelevant context of edits on ASTs, which reduced program lengths by about 50% and increased the overall editing accuracy.

In Section 4.2, we notice that almost half (41.19%) of the ground-truth vulnerability injections only delete line(s). This is natural because the vulnerability fixes, which are the reversal edits of vulnerability injections, tend to add new line(s) in many cases [23]. Graph2Edit captures this pattern and almost always deletes line(s) for the real-world samples, making it achieve 10.03% exactly-match accuracy and 34.93% success rate. While an advantage, this also indicates Graph2Edit’s limited capability to replace code. In comparison, Hoppity has broader capabilities so that it has success cases on different kinds of edits. Nevertheless, it still succeeds more on the deleting line(s) only cases. This indicates that the current neural code editors are more capable of deleting code, because this primitive is simpler (with no need to predict the node type and value). Thus, we suggest to improve the designs of other editing primitives to enhance the ability of editors to add and replace code.

6.4 Model/Algorithm Design

Intuitively, at least for code editing, a more sophisticated model/algorithm (e.g., graph-based, as exemplified in Graph2Edit and Hoppity, which help capture semantic information) is generally expected to outperform a simplified one (e.g., sequence modeling in SequenceR) that ignores semantic information. Yet, in Section 3.1, we observe counter-intuitive results (as summarized in Table 1): The sophisticated designs do not perform much better (with Graph2Edit) or even much worse (with Hoppity) than the (seemingly over-) simplified design (with SequenceR). The reason is that the specific design may also significantly impact the potential. For example, as noted earlier, Hoppity has several critical weaknesses in its design (e.g., mismatch with the nature of the application domain, as exemplified in its lack of an edit primitive for copying a subtree). Our results suggest that its limited accuracy is largely attributed to such major design flaws. In short, a careful design is key for leveraging the potential of a generally more powerful model (architecture) or learning algorithm. Thus, we suggest not to simply adopt a sophisticated model/algorithm assuming it to surely outperform a simplified one, but to compare both.

6.5 Data Quality

In Section 5, we show the impact of different data properties on the accuracy of DL-based vulnerability detectors. Based on the results, we notice that the data quality of the vulnerability samples matters and there are three main aspects we should consider to improve it. Representativeness. Table 9 shows that the synthetic samples are not very helpful or even have a negative impact on the vulnerability detectors for detecting real-world vulnerabilities. In contrast, the DL-generated samples confirmed as vulnerable and realistic, as well as the real-world samples, are much more helpful. The reason is that the synthetic code and vulnerabilities are not representative of those in real-world programs, e.g., in terms of structural complexity and vocabulary diversity. Thus, we suggest that, when building vulnerability datasets, we should aim at realistic samples rather than synthetic ones. As manually curating them may not be a viable path to make the datasets sizable, especially for training DL models, we suggest to develop automated approaches to generating large-scale realistic vulnerability datasets as supported by our study.

Diversity. In Section 4.2, we notice that the neural code editors tend to generate more generic vulnerabilities than domain-specific ones. This distribution difference indicates that the abilities of editors to generate different types of vulnerabilities are imbalanced, which compromises the diversity of the generated data. Thus, we suggest to improve the capability of models on the more difficult types of vulnerabilities so as to improve the diversity.

From our study, an interesting observation is that the graph-based editors are able to generate valid/realistic vulnerable samples that do not match the ground truth (Section 4.1), resulting in some kind of diversity. We also notice that the success of vulnerable injection is not just inadvertently gained by incomplete reversal of
vulnerability-fixing changes to retain the same vulnerability type, but also by additional changes to spawn a new type of vulnerability. Thus, the diversity can be a result of the non-duality between vulnerability injection and vulnerability fixing: to inject vulnerabilities, we do not need complete reversal of all vulnerability-fixing changes. Future work could leverage this non-duality for diverse data generation by decomposing vulnerability-fixing changes and using parts of them combinatorially to inject vulnerabilities.

Noise. In Section 5, we show that even without manual filtering, our DL-generated samples still improve the vulnerability detectors over the baselines. This shows the value of DL-generated samples and thus shed a light on using automatic approaches to generate high-quality vulnerability data. Yet, the improvements by using all these samples are much less than just using the manually selected success samples and the ground-truth samples, because of the noisy nature of the former. Thus, we suggest, for generating high-quality data, to reduce noise in the data, e.g., by using a dynamic analyzer [41, 42] to validate the generated sample as indeed vulnerable.

7 THREATS TO VALIDITY

Internal validity. The major threat to the internal validity of this study lies in the possible errors when we manually review the source code of the techniques. We provide many in-depth technical insights based on the source code we review. Some of the insights are not rigorous proved by experiments or theoretical analysis. To mitigate this problem, we do literature studies on other papers and ensure that our technical insights have related support.

External validity. The major threat to the external validity of our study lies in the datasets and the evaluated techniques we select. The evaluated techniques may not represent the state-of-the-art techniques that are suitable for our software vulnerability data generation task. The datasets we select may not be fairly evaluate the techniques comprehensively, and our insights may not generalize to other datasets. To mitigate the two problems, we set several criteria to select suitable techniques, and we use several different and reliable datasets to validate our insights.

8 RELATED WORK

Many DL-based code editing/transformation techniques were developed. Harer et al. [22] proposed an generative adversarial network (GAN) approach to repair software vulnerabilities without requiring vulnerable and non-vulnerable samples to be paired at training time. Dinella et al. [16] and Yao et al. [59] proposed graph-based models for program editing. Yasunaga et al. [60] proposed a semi-supervised, graph-based model for program repairing based on diagnostic feedback. Chen et al. [11] proposed a NMT-based model to translate buggy code to fixed code. Tarlow et al. [53] used graph2diff neural model to fix build errors in programs. Berabi et al. [6] built a high quality dataset and used it to train a capable text-to-text transformer model to fix software bugs. Yasunaga et al. [61] built an unsupervised model to fix software defects with the help of static analyzers and compilers. We select Sequence2R [11], Graph2Edit [59], and Hoppity [16] for our empirical study, not only because they are the state-of-the-art code editing/transformation techniques, but also they are open-source, standalone, and configurable which well satisfied our task requirements.

Other empirical studies on DL-based code analysis/transformation exist. Kang et al. [26] assessed if the popular model code2vec was able to generalize to the tasks of code comment generation, code author identification, and code clone detection. Ciniselli et al. [12] did an empirical study to evaluate the capability of the BERT model for code completion. They also did an extended study [12] to investigate whether several transformer-based models were capable for code completion task. Rabin et al. [48] compared the capabilities of several neural program models for semantic-preserving code transformations. Tufano et al. [54] assessed the feasibility of using NMT techniques to fix bugs in the wild. Paltenghi and Pradel [44] compare DL-based models of code to human developers. In comparison, we are the first to evaluate the feasibility of using DL-based code editing techniques for software vulnerability data generation.

There have been prior studies on vulnerability datasets. SARD [7] and SATE IV [43] are synthetic databases of over 60,000 vulnerable samples. The CVE/NVD [8] database archives vulnerabilities found in real-world software. The BigVul [17] dataset used in our study was built to include source code associated with known CVEs, including 3,754 pairs of code samples. A few other works [18, 30, 64] built datasets by detecting vulnerability patches. In [63], a special dataset, including only null-pointer-dereference vulnerability samples, was presented. Overall, these datasets are either not affirmatively realistic or in relatively small numbers, as we discuss earlier. SemSeed [45] injects synthetic bugs by imitating real-world bugs, but does not focus specifically on vulnerabilities. In comparison, in this work, we focus on exploring how far we are from automated realistic data generation via neural code editing, while validating the usefulness of such generated datasets.

9 CONCLUSION

We conduct an in-depth exploratory study on realistic vulnerability data generation via neural code editing. Using two vulnerability datasets, we reveal the technical strengths and weaknesses of three state-of-the-art neural code editors. Through extensive empirical and technical analyses, we find key limitations for code editing in general, as well as technical gaps for vulnerability data generation in particular, of these neural editors. On the positive side, we show the greater potential of the graph-based techniques over the sequence-based techniques for generating realistic vulnerable code. We also validate that such generated code samples can indeed boost the accuracy of deep learning-based vulnerability detectors in detecting real-world vulnerabilities. We offer significant insights and actionable suggestions for the design of future neural code editing techniques and generation of realistic, high-quality vulnerability datasets.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their constructive comments. This research was sponsored by the Army Research Office Grant Number W911NF-21-1-0027. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Office or the U.S. Government. This work was also supported by the European Research Council (ERC, grant agreement 851895), and by the German Research Foundation within the ConcSys and DeMoCo projects.
Generating Realistic Vulnerabilities via Neural Code Editing: An Empirical Study

ESEC/FSE ’22, November 14–18, 2022, Singapore, Singapore