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 for both benchmarking existing techniques and developing effective new ones, especially those using data-driven (e.g., machine/deep-learning based) approaches, for software security. Yet such datasets are critically lacking. A promising solution is to generate such datasets by injecting vulnerabilities into real-world normal 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. We also provide insights into the current gaps in these editors (e.g., good at deleting but not for replacing code) and actionable suggestions for addressing them (e.g., designing efficient editing primitives).

CCS CONCEPTS

• **Software and its engineering–AI and software engineering**;

KEYWORDS

Datasets, Data Generation, Data Augmentation, Deep Learning, Software Vulnerability, Vulnerability Detection, Benchmarking

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, 30, 40]) have been proposed, of which the most momentous are those based on deep learning (DL) [20, 25]. Indeed, DL-based software vulnerability (e.g., detection [9, 34, 35, 61], repair [22]) analysis techniques have achieved great successes, reporting performance that often surpassed traditional (e.g., code analysis based) approaches.

However, these promising-looking performance 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 (DL or code-analysis based) 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 performance becomes much lower [9, 61]. Meanwhile, existing comparative studies of the techniques are generally limited to technical discussion and qualitative assessment [2, 28, 48] and/or incomplete comparisons (e.g., just comparing the numbers of vulnerabilities found rather than precision and recall) [3–5, 43].

- **Model training:** *development of new and more effective DL-based techniques.* DL-based techniques for vulnerability analysis have shown great promise. Yet their performance 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 (e.g., detection/discovery [15, 37] and localization [33]) approaches.

Some software vulnerability datasets are available. SARD [7] and SATE IV [42] 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 [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 [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 [18, 29, 60] 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. [59] 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 normal (i.e., non-vulnerable) programs, which are widely available. Given the success of *neural code editors*, 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/algorithm* perspectives. Then, we evaluate these editors' performance 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 performance for 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 performance 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 will be made publicly accessible upon acceptance.

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 performance 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. And 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 these editors' performance. Second, the synthetic nature enables us to experiment with various modifications (e.g., automated code refactoring, variable renaming) of the dataset to assess the editors' generalizability hence gain deeper insights into the potential and gaps. 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., commit logs, 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, 55, 56] 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 [55] and Hoppity [16], two latest graph-based editors, and SequenceR [11], a state-of-the-art sequence-based editor, that (together) 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 performance, it uses the *copy mechanism* to address the unlimited vocabulary problem [46]. 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 [55] 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) [31], and predicts a sequence of edit actions based on the embeddings using a long short-term memory (LSTM) network [55]. 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 the Graph2Edit authors' guidance 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 [54], 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 *succ 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) [53] 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 [47], to obtain the ground-truth edit sequences. For our study, we also use Joern to generate the ASTs (of C samples).

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, <u>*accuracy = #accurate outputs/#inputs*</u>.

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 [52][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 DP algorithm [55]. We finally obtain 7,789 patches covering more than 340 open-source projects where the average number of lines

---

[1]The BigVul used as a basis of our real-world dataset is the collection of C/C++ samples from the CVE/NVD database [39]. As BigVul alone is relatively small, we additionally considered another source (PatchDB) to form our real-world dataset.

Table 1: RQ1: the accuracy on the original synthetic dataset, and that with different treatments for testing samples

Setting	Original	Large Vocab	Complex Code Structure
Hoppity	59.22%	48.65% (17.85%↓)	49.45%(16.50%↓)
Graph2Edit	79.40%	13.13%(83.46%↓)	40.71%(48.73%↓)
SequenceR	72.28%	31.59%(56.28%↓)	2.42%(96.65%↓)

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 editors' performance 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 albeit they have similar designs. One of the reasons is its poor 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 make Hoppity harder to predict the overall edits correctly, significantly limiting its effectiveness.

The greater performance 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.

> The representation of code edit scripts matters. They should be concise, precise, and describe copied code explicitly. Graph-based editors are more effective than sequence-based ones, as graphs model program syntax and semantics better which benefits vulnerability injection.

3.2 Generalizability of the Editing Techniques

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

The first is *Large Vocab*. We evaluate the impact of vocabulary size (#unique tokens) on the editors' performance. As in [58], for each pair of samples in the testing set, we randomly replace the identifiers (function/variable names) with unrelated names (e.g. `int bufsize=0` is changed to `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 a performance 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.

> All the code editing techniques suffer from performance losses with larger vocabulary sizes, which could be mitigated by ensuring *identifier independence*.

The second experiment is *Complex Code Structure*. We evaluate how more complex code structures impact the editors' performance. 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 performance losses in this experiment too, as more complicate code structures mean more a diverse data distribution to learn. Hoppity performs the best with a 49.45% accuracy, because of its use of ASTs and the Graph Isomorphism Network (GIN) [53]. In an AST, each subtree represents a code block/statement. Thus, changing the structure of code only makes sparse changes to the AST. For example, exchanging the if and else bodies only changes the order 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 GIN does.

---

[2]The ability of a learning-based code model to learn the code semantics without being impacted by variations of identifier names.
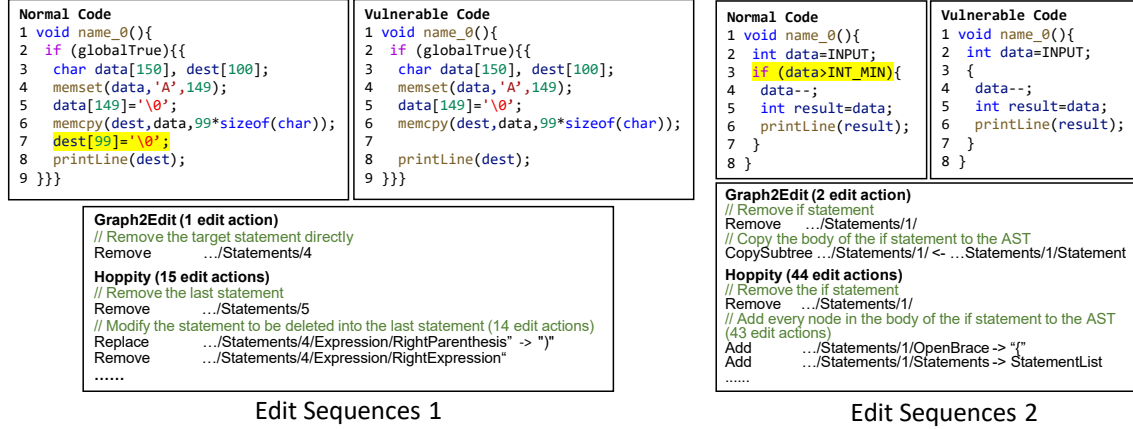
Figure 1: A comparison of the ground-truth edit sequences generated by Graph2Edit and Hoppity.
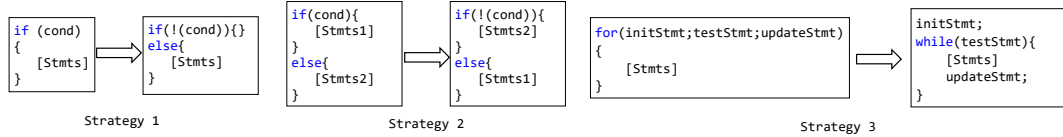

Figure 2: The methods we used to refactor the (synthetic) code.
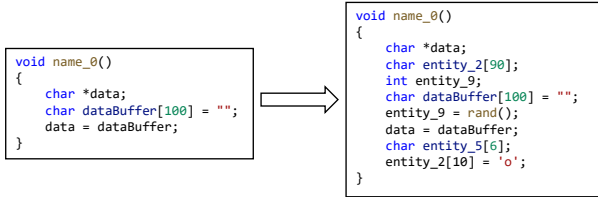

Figure 3: An example of adding unrelated code.

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 performance with more complicated code structures. Sequence-based models are most sensitive to this problem, whereas AST and graph-based models help ensuring generalizability.

3.3 Performance Impact of Key Dataset Factors

We consider three dataset factors key to our study: (1) *edit length* of a testing pair is #edit actions between the pair, (2) *program length* of a testing pair is #tokens in the ground-truth vulnerable sample in the pair, and (3) *pattern frequency* of a testing pair is #sample pairs in the training set that have the same pattern as that of the edits between the pair, where the pattern of each pair is the edit sequence abstracted by only reserving the operator and target node type in each edit action, as illustrated in Figure 4. For edit length and pattern frequency, we refer to the ground-truth edit sequences computed by Graph2Edit as they are optimized and similar to human edits.

Table 2 shows the impact of edit length on the prediction accuracy. The accuracy decreases when the edit lengths increase, for any of the techniques. For Hoppity and Graph2Edit, this is because they predict edit actions iteratively; thus, the greater the edit length, the larger the probability that the overall prediction is incorrect. For SequenceR, while the edit lengths may not directly impact the

Table 2: How does edit length impact accuracy?

Edit Length	<4	4-10	10-40	40-100	>100
Hoppity	77.72%	48.35%	28.08%	0.00%	0.00%
Graph2Edit	94.41%	76.89%	52.71%	3.51%	0.00%
SequenceR	86.43%	69.58%	45.32%	14.04%	0.00%

Table 3: How does pattern frequency impact accuracy?

Pattern Frequency	0	1-10	10-20	20-100	100-500	>500
Hoppity	0.00%	7.92%	9.32%	24.63%	64.87%	86.05%
Graph2Edit	0.00%	29.31%	68.42%	67.74%	71.97%	96.60%
SequenceR	0.00%	25.00%	31.36%	64.93%	70.26%	91.99%

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 frequencies increase. This can be explained by the nature of the gradient descent (GD) algorithm [45] used by these techniques 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 the extreme case: the techniques cannot predict correctly on the 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 performance of all the techniques generally decreases significantly when the edit length or program length is much greater, or when the pattern frequency is lower. This indicates the limitation of the techniques in the complex cases of vulnerability injection.
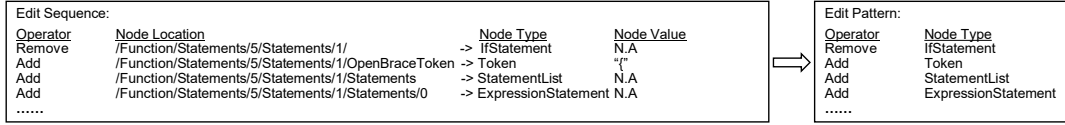
```
Edit Sequence:
Operator    Node Location                                          Node Type            Node Value        Edit Pattern:
Remove      /Function/Statements/5/Statements/1/                   -> IfStatement        N.A              Operator    Node Type
Add         /Function/Statements/5/Statements/1/OpenBraceToken     -> Token              "{"              Remove      IfStatement
Add         /Function/Statements/5/Statements/1/Statements         -> StatementList      N.A              Add         Token
Add         /Function/Statements/5/Statements/1/Statements/0       -> ExpressionStatement N.A             Add         StatementList
......                                                                                                     Add         ExpressionStatement
                                                                                                          ......
```

Figure 4: An example of getting the edit pattern from an edit sequence.

Table 4: How does program length impact accuracy?

# of tokens in programs	<100	100-125	125-150	150-175	175-200	>200
Hoppity	55.96%	56.82%	70.18%	68.45%	58.06%	34.38%
Graph2Edit	84.44%	79.55%	46.84%	78.64%	73.12%	64.46%
SequenceR	76.49%	76.62%	72.28%	71.84%	60.22%	46.88%

Table 5: RQ2: the accuracy on the real-world dataset

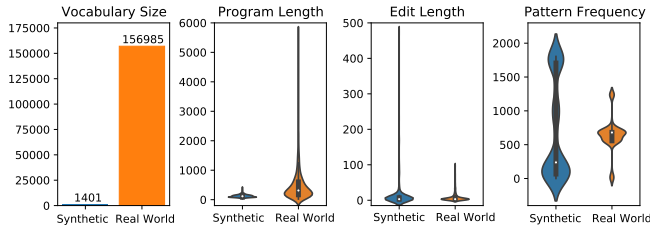| Editor	Accuracy (Exactly-Matched)	Success Rate
Hoppity	1.41%	9.09%
Graph2Edit	10.03%	34.93%
SequenceR	0.00%	0.00%

Figure 5: The synthetic versus real-world datasets.

4 RQ2: REALISTIC DATA GENERATION

To answer this question, 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—we intend to have more samples for testing so that more data can be used in RQ3.

4.1 Results on the Real-World Dataset

Table 5 (2nd column) shows the editors' (*exactly-match*) accuracy on our real-world dataset. The results are much worse than the ones 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 hence make the experiments infeasible. For RQ1, we have discussed the impacts of these factors with the synthetic dataset. With the real-world dataset, these impacts are much greater, 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 poor preprocessing that generates redundant ground-truth edit sequences. Yet its use of AST, identifier independence, and 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) it has no sense of code syntax and semantics; and (3) it 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 allow 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, sequence-based editors may not be promising approaches for generating vulnerabilities.

> All the techniques have bad 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 performance, indicating their gaps for realistic vulnerability generation.

On the other hand, the way the vulnerabilities are injected may not be unique for a given normal sample. It is possible that a vulnerability has already been injected, but resulting code does not exactly match the ground truth. Thus, we manually check each DL-generated code sample that does not match its ground truth. If we find that such a code sample has vulnerabilities, we still mark it as a *success* sample. Table 5 (3rd column) shows the *success* rates on the real-world dataset, where a success rate is the percentage of *exactly-matched* samples *and* other *success* samples in the testing set. For both Graph2Edit and Hoppity, the *success* rate is notably higher than the *exactly-matched* accuracy. In particular, Graph2Edit achieves a much greater success rate (34.93%), suggesting its better potential for vulnerability data generation.

Given the best exact-matching accuracy of 10.03%, we had to manually inspect 90% of the generated samples in order to identify all the non-exact-matching success cases. Thus, this (Graph2Edit-based) generation approach is not yet ready for practical adoption in generating realistic vulnerabilities at scale. Yet recall that our paper is an empirical study aimed to understand the potential and gaps regarding such approaches, rather than developing such an approach itself. The goal of manually inspecting 90% of the generated samples is only to assess the potential of the neural code editors for generating vulnerable samples that do not exactly match ground truth, and to help show the potential of the generated samples for improving vulnerability detectors (as we show in the results for RQ3). Users do not have to perform this inspection—as shown in Table 9, even the generated samples without inspection/filtering can improve the detectors' performance most of the time, showing the practicality of neural-code-editing-based vulnerability generation.

> Graph-based techniques still successfully generate many vulnerable samples including the ones that do not exactly match the ground truths, indicating their potential for generating realistic vulnerabilities.

Table 6: How do the kinds of edits (add/remove/replace lines) done by the models and those in the ground truth differ?

Tool	Edits	% Add line(s) only	% Delete line(s) only	% Replace line(s)
Ground truth	All	1.07%	41.19%	57.74%
Graph2Edit	All	0.13%	99.66%	0.21%
	Success, exactly-match	0.00%	100%	0.00%
	Success, not exactly-match	0.00%	100%	0.00%
Hoppity	All	1.41%	14.88%	83.71%
	Success, exactly-match	3.03%	57.58%	39.39%
	Success, not exactly-match	4.47%	59.78%	35.75%

4.2 Case Studies

Based on our manually inspection on the DL-generated samples, we notice some patterns in the generation process. Thus, we quantitatively investigate two questions on the neural code editors' behaviours. Since SequenceR could not generate (syntactically) valid samples in our testing (Section 4.1), we only do the case studies 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 are good at and tend to do. To be consistent across Graph2Edit and Hoppity, we use the *diff* tool [38] to get the lines added/deleted between the DL-generated/ground-truth vulnerable samples and the normal ones. We then examine how often the models *add line(s) only*, *delete line(s) only*, or do both (*replace line(s)*), versus the ground truth (line-level) edits.

Table 6 shows the results. Of the ground-truth edits, more than half (57.74%) are replacing line(s) and 41.19% 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%), covering the entire set of *success* ones. 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 the advantage of its *replacing node value* and *replacing node type* operators and thus reduces 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 ">="), which is what Hoppity is good at.

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 the 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 *adding a node* and *replacing a node value/type* operators, *deleting a*


```c
// bigvul_linux_CVE-2017-16534_CWE-119_linux_CVE-2017-16534_CWE-119_good.c
int cdc_parse_cdc_header(struct usb_cdc_parsed_header *hdr,
    struct usb_interface *intf, u8 *buffer, int buflen){
    ......
    unsigned int elength;
    int cnt = 0;
    while (buflen > 0){
    elength = buffer[0];
    Generic vulnerability:
    removing the checking of buffer may cause memory safety issue (CWE-119),
    which is a vulnerability for software in all the domains.
    if ((buflen < elength) || (elength < 3)){
        dev_err(&intf->dev, "invalid descriptor buffer length\n");
        break;
    }
    ......
    buflen -= elength;
    buffer += elength;
    }
    ......
}
```

```c
// bigvul_libuv_CVE-2015-0278_CWE-264_libuv_CVE-2015-0278_CWE-264_good.c
static void uv__process_child_init(const uv_process_options_t *options,
    int stdio_count, int (*pipes)[2], int error_fd){
    ......
    Domain specific vulnerability:
    not dropping the user/group priviliage may cause permission issues (CWE-264),
    which is a vulnerability depending on the software domain and usage.
    if (options->flags & (UV_PROCESS_SETUID | UV_PROCESS_SETGID))
        SAVE_ERRNO(setgroups(0, NULL));
    execvp(options->file, options->args);
    uv__write_int(error_fd, -errno);
    perror("execvp()");
    _exit(127);
}
```

Figure 6: Generic versus domain-specific vulnerabilities.

node can delete a subtree by simply deleting the root node while adding/replacing a subtree needs to add/replace the nodes one by one. Also, *deleting a node* operator only needs to know the node location, but not the value and node type as other operators do.

While *adding line(s)* edits are rare in the ground truth and all the Hoppity's edits, we notice that there are still success cases for such edits on Hoppity. This indicates that Hoppity potentially has a broader set of capabilities for vulnerability injection.

> 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.

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 this diversity/similarity is to differentiate the vulnerabilities between generic and domain-specific. We refer to as a *generic vulnerability* a bug that may cause security issues in any domains 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.

We randomly sample 200 ground-truth vulnerable samples from the entire real-world dataset and 200 DL-generated samples from the exactly-match cases. Then, we manually identify whether the vulnerabilities are generic or domain-specific. We do not check the *success but not-exactly-match* cases because these cases are

selected by human 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, which is also what it is good at as we found for Q1. 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.

Note that we built the underlying (real-world) dataset based on BigVul and PatchDB without considering the differentiation between generic versus domain-specific classes. These classes were only manually identified *after* the dataset was built. We chose this dataset because the two sources (BigVul, PatchDB) cover more (340+) real-world projects than other ones that we know of (e.g., the ReVeal and Devign datasets each covering 2 projects only).

> 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. The manual inspection was performed by the first author, who has 4 years of experience in software engineering and security.

We then shuffle these 40 samples for each participant, who is asked to rate their confidence (out of 4 levels where 4 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.
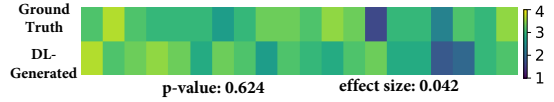

Figure 7: The average realism confidence levels of the 20 ground-truth samples versus the 20 DL-generated samples.

Table 7: Are the editors better for generic vulnerabilities?

Case	% domain specific
Ground-truth samples	47.50%
Exactly-match samples by Graph2Edit	43.50%
Exactly-match samples by Hoppity	15.00%

Table 8: Vulnerability detection datasets

Dataset	#Vulnerable Samples	#Normal Samples	%Vulnerable
Devign	10,051	11,012	47.71%
ReVeal	1,658	16,511	9.12%
Xen	531	7,436	6.67%

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 [51] 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.

> By doing a user study followed by statistical analyses, we find that our DL-generated success samples are realistic.

5 RQ3: USEFULNESS OF GENERATED DATA

In this RQ, we investigate whether our DL-generated vulnerable samples help improve DL-based vulnerability detection, which tells a given sample is vulnerable or not. We choose two state-of-the-art detectors, Devign [61] and ReVeal [9], as they are considered the most effective such detectors for C so far [9, 32]. 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 where 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 [36] 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: the training set is **ReVeal** and the testing set is **Devign**.

We use these three settings against the two detectors as baselines. Then, we improve their training set by adding the $2,333 \times 34.93\% = 815$ 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 [36] (other than those already included in the **Xen** dataset above) such that the balance does not change.

Table 9: RQ3: Does the DL-generated samples help improve the DL-based vulnerability detectors?

Tool	Setting	Metric	Baseline	Improved	Synthetic	Ground Truth	All Generated
Design	**Reproduction:**	Precision	10.75%	9.67%↑	9.95%↑	8.56%↑	3.35%↑
	Training: Devign	Recall	38.78%	37.26%↑	11.76%↑	56.88%↑	57.86%↑
	Testing: ReVeal	**F1**	**16.83%**	**14.74%↑**	**10.40%↑**	**16.34%↑**	**11.76%↑**
	Partial Replication:	Precision	8.73%	16.04%↑	-10.31%↑	30.81%↑	5.04%↑
	Training: Devign	Recall	37.48%	102.48%↑	-1.52%↑	50.24%↑	26.12%↑
	Testing: Xen	**F1**	**14.16%**	**26.20%↑**	**-8.76%↑**	**34.11%↑**	**8.47%↑**
	Full Replication:	Precision	55.56%	-3.67%↑	0.92%↑	1.64%↑	7.61%↑
	Training: ReVeal	Recall	2.76%	38.04%↑	8.70%↑	74.28%↑	5.43%↑
	Testing: Devign	**F1**	**5.25%**	**35.43%↑**	**8.57%↑**	**68.76%↑**	**5.52%↑**
ReVeal	**Reproduction:**	Precision	11.24%	9.79%↑	2.94%↑	22.06%↑	-1.42%↑
	Training: Devign	Recall	68.82%	13.27%↑	9.39%↑	-8.28%↑	22.65%↑
	Testing: ReVeal	**F1**	**19.31%**	**10.36%↑**	**3.94%↑**	**16.68%↑**	**1.45%↑**
	Partial Replication:	Precision	6.21%	34.94%↑	12.88%↑	40.90%↑	34.62%↑
	Training: Devign	Recall	29.94%	210.09%↑	-49.06%↑	184.94%↑	204.44%↑
	Testing: Xen	**F1**	**10.28%**	**49.51%↑**	**-6.61%↑**	**54.38%↑**	**48.93%↑**
	Full Replication:	Precision	53.62%	5.86%↑	-2.93%↑	-2.24%↑	-0.71%↑
	Training: ReVeal	Recall	22.67%	20.78%↑	-60.87%↑	36.48%↑	19.89%↑
	Testing: Devign	**F1**	**31.86%**	**15.94%↑**	**-52.42%↑**	**22.16%↑**	**12.96%↑**

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 (e.g., in reproduction, Devign's F1 improves from 16.83%, by 14.74%, to 19.31%). 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 replication. This indicates that the DL-generated vulnerable samples have very good potentials—they are indeed useful in boosting the effectiveness of DL-based vulnerability analysis techniques.

To show that the DL-generated data is more useful than the synthetic, we replace the 815 DL-generated samples with 815 vulnerable samples from the synthetic dataset used in RQ1. Then, we redo the experiments with no other changes. Table 9 column `Synthetic` shows the results: the overall improvements are much less than those in the `Improved` experiments (e.g., 8.57% versus 35.43% in Devign's F1 in full replication). In several cases, the effectiveness even decreases (e.g., by 52.42% in ReVeal's F1 in full replication). This indicates that the quality of the synthetic samples is lower than the DL-generated ones which are more realistic.

We also compare the DL-generated vulnerable samples with the ground-truth ones. From these samples, we replace the *success but not-exactly-match* ones with their ground truths. As Table 9 column `Ground Truth` shows, this brings even greater improvements (e.g., 16.68% versus 10.36% in ReVeal's F1 in reproduction). This indicates that while the DL-generated samples are realistic and useful, they still have gaps in these regards from real-world vulnerabilities.

Finally, we try to use all the 2,333 DL-generated samples. To be consistent with the previous experiments (keeping the data balance and #added samples the same), we undersample the 2,333 by randomly removing samples until 815 is left. Then, we add these 815 to the baseline training sets. As Table 9 column `All Generated` shows, in any setting, the improvements are less than the ones in the `Improved` experiments in terms of F1 (e.g., 1.45% versus 10.36% with ReVeal in reproduction) because of the noisy samples. However, the augmentation still brings improvements over the `Baseline`, and in most cases the `Synthetic`, experiments. This indicates that the DL-generated samples have the potential to improve the vulnerability datasets even without manual filtering.

On a side note, generally the numbers of generated samples added do affect the improvement of the vulnerability detectors achieved by taking those additional samples. In our preliminary experiments, the larger the number of realistic samples added, the greater the improvement seen by the detectors.

> Despite gaps from real-world data, DL-generated samples have great (and greater-than-synthetic-ones) potentials for boosting DL-based vulnerability analysis techniques.

6 DISCUSSION

We further discuss what makes effective technical (neural network model and learning algorithm) design for neural code editing in general and realistic data generation in particular.

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 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 model 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 overflow). 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 code itself.

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's best performance among the three 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 AST has richer, easily-accessible information (syntactic structure) than text sequences—important for vulnerable code generation. The GNN used also helps—it only takes a few steps of convolution hence avoids overfitting by restricting message-passing on local structures. While LSTM in SequenceR allows forgetting and dynamically adjusting its memory, the model can still learn noisy long-distance correlations between tokens.

Yet AST does 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 on the AST. This contributes to Hoppity's poor performance because the common traits of code editing is 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 performance (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 poor 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 editors' capability for adding/replacing 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 is able to capture semantic code information) is generally expected to outperform a simplified one (e.g., sequence modeling like 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 poor performance 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, even for a task for which the sophisticated one should potentially outperform.

6.5 Data Quality

In Section 5, we show the impacts of different data on the performance of the 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 negative impacts 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, vocabulary diversity, etc.). 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 editors' capabilities of generating different types of vulnerabilities are imbalanced, which compromises the diversity of the generated data. Thus, we suggest to improve the models' capability 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 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. We hence suggest to 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 noises (wrong labels) in the former. Thus, we suggest, for generating high-quality data, to reduce noise in the data (e.g., by using a dynamic analyzer [40, 41] 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 fair enough to evaluate the techniques comprehensively. The insights obtained may not be generalized to other datasets. To mitigate the two problems. We set several criteria to better select the suitable techniques. We use several different and reliable datasets to validate many of our insights for several times.

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. [55] proposed graph-based models for program editing. Yasunaga et al. [56] 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. [49] 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. [57] built an unsupervised model to fix software defects with the help of static analyzers and compilers. We select SequenceR [11],

Graph2Edit [55], 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. [44] compared the capabilities of several neural program models for semantic-preserving code transformations. Tufano et al. [50] assessed the feasibility of using NMT techniques to fix bugs in the wild. 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 [42] 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, 60] built datasets by detecting vulnerability patches. In [59], 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. 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 design flaws for code editing in general, as well as technical gaps for vulnerability data generation in particular, of these neural editors. On the other hand, we show the greater potential of the graph-based techniques over the sequence-based one for generating realistic vulnerable code. We also validate that such generated code samples can indeed boost the performance 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.

REFERENCES

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).

[2] Richard Amankwah, Patrick Kwaku Kudjo, and Samuel Yeboah Antwi. 2017. Evaluation of software vulnerability detection methods and tools: a review. *International Journal of Computer Applications* 169, 8 (2017), 22–7.

[3] Nuno Antunes and Marco Vieira. 2009. Comparing the effectiveness of penetration testing and static code analysis on the detection of SQL injection vulnerabilities in web services. In *Pacific Rim International Symposium on Dependable Computing*. 301–306.

[4] Andrew Austin, Casper Holmgreen, and Laurie Williams. 2013. A comparison of the efficiency and effectiveness of vulnerability discovery techniques. *Information and Software Technology* 55, 7 (2013), 1279–1288.

[5] Andrew Austin and Laurie Williams. 2011. One technique is not enough: A comparison of vulnerability discovery techniques. In *International Symposium on Empirical Software Engineering and Measurement*. IEEE, 97–106.

[6] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. Tfix: Learning to fix coding errors with a text-to-text transformer. In *International Conference on Machine Learning*. PMLR, 780–791.

[7] Paul E Black et al. 2017. SARD: A Software Assurance Reference Dataset. https://samate.nist.gov/SARD/. In *Anonymous Cybersecurity Innovation Forum.()*.

[8] Harold Booth, Doug Rike, Gregory A Witte, et al. 2013. The national vulnerability database (nvd): Overview. (2013).

[9] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).

[10] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2021. Neural Transfer Learning for Repairing Security Vulnerabilities in C Code. *arXiv preprint arXiv:2104.08308* (2021).

[11] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* (2019).

[12] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2021. An Empirical Study on the Usage of Transformer Models for Code Completion. *arXiv preprint arXiv:2108.01585* (2021).

[13] Fırat Civaner. 2020. Real-Life Software Security Vulnerabilities And What You Can Do To Stay Safe. https://hackernoon.com/how-software-security-vulnerabilities-work-and-what-you-can-do-to-stay-safe-c9596d993581.

[14] Norman Cliff. 2014. *Ordinal methods for behavioral data analysis*. Psychology Press.

[15] Hoa Khanh Dam, Truyen Tran, Trang Thi Minh Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. 2018. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering* (2018).

[16] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations (ICLR)*.

[17] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.

[18] Therese Fehrer, Rocío Cabrera Lozoya, Antonino Sabetta, Dario Di Nucci, and Damian A Tamburri. 2021. Detecting Security Fixes in Open-Source Repositories using Static Code Analyzers. *arXiv preprint arXiv:2105.03346* (2021).

[19] Xiaoqin Fu and Haipeng Cai. 2021. FlowDist:Multi-Staged Refinement-Based Dynamic Information Flow Analysis for Distributed Software Systems. In *30th USENIX Security Symposium (USENIX Security 21)*. 2093–2110.

[20] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. 2017. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *Comput. Surveys* 50, 4 (2017), 1–36.

[21] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. 2016. Incorporating Copying Mechanism in Sequence-to-Sequence Learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 1631–1640. https://doi.org/10.18653/v1/P16-1154

[22] Jacob Harer, Onur Ozdemir, Tomo Lazovich, Christopher Reale, Rebecca Russell, Louis Kim, et al. 2018. Learning to repair software vulnerabilities with generative adversarial networks. *Advances in Neural Information Processing Systems* 31 (2018).

[23] Emanuele Iannone, Roberta Guadagni, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. 2022. The Secret Life of Software Vulnerabilities: A Large-Scale Empirical Study. *IEEE Transactions on Software Engineering* (2022).

[24] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173.

[25] Gong Jie, Kuang Xiao-Hui, and Liu Qiang. 2016. Survey on software vulnerability analysis method based on machine learning. In *IEEE First International Conference on Data Science in Cyberspace*. IEEE, 642–647.

[26] Hong Jin Kang, Tegawendé F Bissyandé, and David Lo. 2019. Assessing the generalizability of code2vec token embeddings. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1–12.

[27] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M Rush. 2017. Opennmt: Open-source toolkit for neural machine translation. *arXiv preprint arXiv:1701.02810* (2017).

[28] Peng Li and Baojiang Cui. 2010. A comparative study on software vulnerability static analysis techniques and tools. In *International Conference on Information Theory and Information Security*. 521–524.

[29] Wen Li, Li Li, and Haipeng Cai. 2022. On the Vulnerability Proneness of Multilingual Code. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[30] Wen Li, Jiang Ming, Xiapu Luo, and Haipeng Cai. 2022. PolyCruise: A Cross-Language Dynamic Information Flow Analysis. In *31st USENIX Security Symposium (USENIX Security 22)*.

[31] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).

[32] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 292–303.

[33] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. 2020. VulDeeLocator: A Deep Learning-based Fine-grained Vulnerability Detector. *arXiv preprint arXiv:2001.02350* (2020).

[34] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. SySeVR: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* (2021).

[35] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).

[36] Guanjun Lin, Wei Xiao, Jun Zhang, and Yang Xiang. 2019. Deep learning-based vulnerable function detection: A benchmark. In *International Conference on Information and Communications Security*. Springer, 219–232.

[37] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Olivier De Vel, Paul Montague, and Yang Xiang. 2019. Software Vulnerability Discovery via Learning Multi-domain Knowledge Bases. *IEEE Transactions on Dependable and Secure Computing* (2019).

[38] Linux Utilities. 2022. The Linux Diff tool. https://www.man7.org/linux/man-pages/man1/diff.1.html.

[39] National Institute of Standards and Technology (NIST). 2022. National Vulnerability Database (NVD). https://nvd.nist.gov.

[40] Yu Nong and Haipeng Cai. 2020. A preliminary study on open-source memory vulnerability detectors. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 557–561.

[41] Yu Nong, Haipeng Cai, Pengfei Ye, Li Li, and Feng Chen. 2021. Evaluating and comparing memory error vulnerability detectors. *Information and Software Technology* 137 (2021), 106614.

[42] Vadim Okun, Aurelien Delaitre, Paul E Black, et al. 2013. Report on the static analysis tool exposition (sate) iv. *NIST Special Publication* 500 (2013), 297.

[43] Davide Pozza, Riccardo Sisto, Luca Durante, and Adriano Valenzano. 2006. Comparing lexical analysis tools for buffer overflow detection in network software. In *1st International Conference on Communication Systems Software & Middleware*. IEEE, 1–7.

[44] Md Rafiqul Islam Rabin, Nghi DQ Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of Neural Program Models with respect to semantic-preserving program transformations. *Information and Software Technology* 135 (2021), 106552.

[45] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).

[46] Abigail See, Peter J Liu, and Christopher D Manning. 2017. Get to the point: Summarization with pointer-generator networks. *arXiv preprint arXiv:1704.04368* (2017).

[47] Shape Security, Inc. 2022. Shift Parser. https://shift-ast.org/parser.html.

[48] Peter Silberman and Richard Johnson. 2004. A comparison of buffer overflow prevention implementations and weaknesses. *IDEFENSE, August* (2004).

[49] Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. 2020. Learning to fix build errors with graph2diff neural networks. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. 19–20.

[50] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 4 (2019), 1–29.

[51] Ronald E Walpole, Raymond H Myers, Sharon L Myers, and Keying Ye. 1993. *Probability and statistics for engineers and scientists*. Vol. 5. Macmillan New York.

[52] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. 2021. Patchdb: A large-scale security patch dataset. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 149–160.

[53] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations*. https://openreview.net/forum?id=ryGs6iA5Km

[54] Fabian Yamaguchi. 2022. A platform for robust analysis of C/C++ code. https://joern.readthedocs.io/en/latest/installation.html.

[55] Ziyu Yao, Frank F Xu, Pengcheng Yin, Huan Sun, and Graham Neubig. 2021. Learning Structural Edits via Incremental Tree Transformations. *arXiv preprint arXiv:2101.12087* (2021).

[56] Michihiro Yasunaga and Percy Liang. 2020. Graph-based, self-supervised program repair from diagnostic feedback. In *International Conference on Machine Learning*. PMLR, 10799–10808.

[57] Michihiro Yasunaga and Percy Liang. 2021. Break-It-Fix-It: Unsupervised Learning for Program Repair. *arXiv preprint arXiv:2106.06600* (2021).

[58] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[59] Shasha Zhang. 2021. A Framework of Vulnerable Code Dataset Generation by Open-Source Injection. In *2021 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*. IEEE, 1099–1103.

[60] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2a: A dataset built for ai-based vulnerability detection methods using differential analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 111–120.

[61] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *arXiv preprint arXiv:1909.03496* (2019).