VGX: Large-Scale Sample Generation for Boosting Learning-Based Software Vulnerability Analyses

Yu Nong
Washington State University
yu.nong@wsu.edu

Richard Fang
Washington State University
richardfang2005@gmail.com

Guangbei Yi
Washington State University
guangbei.yi@wsu.edu

Kunsong Zhao
Hong Kong Polytechnic University
kunsong.zhao@connect.polyu.hk

Xiapu Luo
Hong Kong Polytechnic University
csxluo@comp.polyu.edu.hk

Feng Chen
The University of Texas at Dallas
feng.chen@utdallas.edu

Haipeng Cai*
Washington State University
haipeng.cai@wsu.edu

ABSTRACT

Accompanying the successes of learning-based defensive software vulnerability analyses is the lack of large and quality sets of labeled vulnerable program samples, which impedes further advancement of those defenses. Existing automated sample generation approaches have shown potentials yet still fall short of practical expectations due to the high noise in the generated samples. This paper proposes VGX, a new technique aimed for *large-scale* generation of *high-quality* vulnerability datasets. Given a normal program, VGX identifies the code contexts in which vulnerabilities can be injected, using a customized Transformer featured with a new value-flow-based position encoding and pre-trained against new objectives particularly for learning code structure and context. Then, VGX materializes vulnerability-injection code editing in the identified contexts using patterns of such edits obtained from both historical fixes and human knowledge about real-world vulnerabilities.

Compared to four state-of-the-art (SOTA) (i.e., pattern-, Transformer-, GNN-, and pattern+Transformer-based) baselines, VGX achieved 99.09-890.06% higher F1 and 22.45%-328.47% higher label accuracy. For in-the-wild sample production, VGX generated 150,392 vulnerable samples, from which we randomly chose 10% to assess how much these samples help vulnerability detection, localization, and repair. Our results show SOTA techniques for these three application tasks achieved 19.15–330.80% higher F1, 12.86–19.31% higher top-10 accuracy, and 85.02–99.30% higher top-50 accuracy, respectively, by adding those samples to their original training data. These samples also helped a SOTA vulnerability detector discover 13 more real-world vulnerabilities (CVEs) in critical systems (e.g., Linux kernel) that would be missed by the original model.

*Haipeng Cai is the corresponding author.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639116

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

vulnerability dataset, vulnerability injection, data quality, vulnerability analysis, deep learning, program generation

ACM Reference Format:
Yu Nong, Richard Fang, Guangbei Yi, Kunsong Zhao, Xiapu Luo, Feng Chen, and Haipeng Cai. 2024. VGX: Large-Scale Sample Generation for Boosting Learning-Based Software Vulnerability Analyses. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639116

1 INTRODUCTION

Modern software is widely afflicted by security vulnerabilities [30, 31, 40, 41], which are increasingly consequential [9, 42]. Thus, it is crucial to develop effective defenses against vulnerabilities, for which data-driven, especially deep-learning-based methods have demonstrated tremendous potential, including vulnerability detection [64, 70–72], localization [28, 35, 44], and repair [20, 29, 34].

Yet accompanying the rising momentum of deep learning (DL) in software assurance is the glaring lack of quality training data. In fact, this problem has drawn growing attention from both academia and industry in recent years [10–12, 18, 19, 52, 53, 68]. Manual curation of such datasets is intuitively tedious hence clearly undesirable, and it can at best produce relatively small datasets hence unscalable [15, 26, 50, 62]. Therefore, *automatically* generating vulnerable samples at large scale and with high quality is of paramount significance.

In response, a few data generation techniques have been developed, including neural code editing [18, 25, 65, 66] and control-flow-based code stitching [49]. However, the former suffers a major chicken-egg problem—training the neural code editor requires a large set of quality vulnerable samples which are what we are lacking. The latter, by inserting artificial vulnerable code patterns to real-world code, results in unrealistic vulnerabilities which only have limited usage. Other, even earlier approaches [38, 39, 68, 69], including adaptable ones originally designed for program bug repair [14, 25] are subject to even greater limitations (e.g., higher noise [66] and lower coverage of vulnerability classes [67]). Lately,

VulGen [55] made good progress; yet it still suffers from high noise in the generated data due to its low generation accuracy, as well as overfitting to the seed vulnerable samples it learns from.

In this paper, aiming at *large-scale* generation of *high-quality* vulnerable program samples, we developed an advanced vulnerability injection technique, named VGX (short for **V**ulnerability **G**eneration e**X**panded). VGX combines Step (1) *semantics-aware contextualization*, which identifies the context of vulnerability-injection code edits, with Step (2) *human-knowledge-enhanced edit pattern formation*, using the materialized contextualized code edits.

In Step **(1)**, VGX builds a Transformer-based contextualization model by designing a novel attention mechanism with absolute and relative position encoding both based on value flow relationships among code tokens, and then pre-training the customized Transformer on a large code corpus, followed by fine-tuning it on *an existing (task-specific) dataset* of vulnerability-introducing code locations. To benefit more from the pre-trained model, VGX also introduces new pre-training objectives explicitly geared to our particular task (of contextualizing injection edits).

In Step **(2)**, VGX starts with extracting vulnerability-introducing code edit patterns from the same task-specific dataset and then enriches the extracted patterns with manually identified, different patterns, followed by diversifying the enriched patterns through manually derived pattern mutation rules. Both the manual pattern and mutation rule definitions are based on human knowledge about existing real-world vulnerabilities garnered from CWE/CVEs on NVD [50] and bug/issue reports on GitHub.

The semantics-informed design in Step (1) helps VGX achieve effective contextualization of vulnerability-injecting code edits, while the human knowledge incorporation in Step (2) helps VGX overcome potential overfitting to the small task-specific dataset. In addition, we also (1) pre-train the programming language model on the ASTs of the input code corpus in order to better learn syntactic and structural information in programs and (2) improve the training with data augmentation (through semantics-preserving code refactoring), as inspired by earlier works [49, 51].

To assess VGX, we pre-trained its contextualization model on 1,214K C functions and fine-tuned it on 7K real-world vulnerability-fixing samples augmented by 156K of their refactored versions. From these 7K samples along with manual pattern refinement and diversification, we obtained 604 vulnerability-introducing edit patterns. We then conducted five sets of experiments.

In the **first** set, we evaluated the accuracy of VGX versus four state-of-the-art baselines (VulGen [55], CodeT5 [63], Getafix [14], and Graph2Edit [65]) against 775 testing (normal) samples with ground-truth (vulnerable versions). VGX achieves 59.46% precision, 22.71% recall, and 32.87% F1 (239.77%-1173.23%, 44.28%-780.23%, 99.09%-890.06% higher than the baselines), when considering as true positives the generated vulnerable samples exactly matching ground truths. When counting all of the generated samples that are indeed vulnerable (i.e., not exactly matching the ground truth but still exploitable) as success cases, VGX achieved a 93.02% success rate, 22.45%-328.47% higher than the baselines. In the **second** set of experiments, we showed that each of the novel design elements of VGX (especially the value-flow-based semantics-aware attention mechanism and human-knowledge-based enhancement of edit patterns) played a significant role in its overall performance merits.

In the **third** experiment set, we deployed VGX in the wild on 738K samples, hence producing 150K vulnerable samples in 50 hours 48 minutes with a 90.13% success rate, 118.07% more accurate than VulGen—the best baseline. Then, in the **fourth** set, we augmented the training sets of SOTA defensive vulnerability analyses with 10% of those 150K samples, considering the scalability of the analysis tools. In this way, VGX helped improve (1) (function-level) vulnerability detection by 19.15%-330.80% in terms of F1; (2) (line-level) vulnerability localization by 12.86%-19.31% in terms of top-10 accuracy; and (3) vulnerability repair by 85.02%-99.30% in terms of top-50 accuracy. This is 4.19%-266.42% higher than adding VulGen's generated and existing synthetic samples. Finally, in the **fifth** set of experiments, we applied the augmented version of a SOTA vulnerability detector to 71 latest CVEs in real-world projects and found 62, including 13 that would be missed by the original model.

While we currently implement hence evaluate VGX for C programs, our VGX approach/methodology is not limited to a particular programming language. The proposed contextualization and pattern mining techniques can be readily adapted for a different language, if given the AST and value flow parser for that language. The step of enhancing vulnerability-injection edit patterns based on human knowledge also follows a principled procedure, which can be reasonably replicated for other languages as well.

In summary, this paper contributes the following:

- A novel design for injection-based vulnerable sample generation that leverages *code structure- and context-aware source code pre-training*, *code semantics-aware Transformer attention*, data augmentation, and *human knowledge* to overcome key limitations of existing peer approaches.
- An implementation and evaluation of the design that shows its *significantly superior performance over four SOTA baselines*, including the latest, best-performing peer technique.
- A *large and quality set of 150K vulnerable samples* produced using the implementation, which comes with corresponding normal code, ground-truth vulnerability and locations, and high label accuracy, hence ready for public use.
- Empirical evidences that show the practical usefulness of the generated dataset in terms of the *substantial improvement it brought to vulnerability detection, localization, and repair*, as well as its ability to enable the discovery of real-world vulnerabilities (CVEs) that original models miss.

Open science. Source code of VGX along with all of the experimental and resulting datasets are available here.

2 BACKGROUND AND MOTIVATION

To automatically generate vulnerable samples, an intuitive idea is to (1) learn the patterns of known code edits that introduce vulnerabilities (i.e., *vulnerability-introducing edits*) and then (2) apply to a normal program the patterns that are compatible with it, resulting in a vulnerable version of the program. Such an approach has been demonstrated to be meritorious in earlier works [14, 55, 66]. Most recently, Nong et al. proposed VulGen [55], an injection-based vulnerable code generation technique that also leverages this strategy. Besides the learned patterns, it utilizes a Transformer-based localization model to locate *where* to inject vulnerabilities. While an

Figure 1: A motivating example on vulnerability injection.

important step, VulGen still falls short of large-scale high-quality vulnerability data generation due to the following limitations:

Limitation ①: The learned vulnerability-injection code edit patterns are limited to those seen in the small vulnerability dataset. VulGen mines such patterns from existing vulnerability-introducing code edits. Yet since available datasets of such edits are (even collectively) limited, the patterns mined are a small subset of the extant knowledge about how real-world vulnerabilities can be introduced to software (as embodied in extant vulnerability documentation such as CWE/CVEs in NVD and issues/bug reports on GitHub). For example, in Figure 1, the statements marked gray at Lines 5-6 and 7-8 can be injected "null-pointer-dereference" and "buffer-overflow" vulnerabilities, respectively, by deleting the `if`-statements. However, patterns as such may be *too general* such that the condition and the return values in Lines 5 and 6 are not specified, or *too specific* such that all the tokens for the condition in Line 7 are specified. Only the tokens "NULL", "ENOMEM", "EINVAL" are crucial for vulnerability injection in this example, but the respective patterns are difficult to mine due to the lack of available code samples covering them. Thus, *to leverage their full potential, the patterns need to better cover the extant knowledge, as can be achieved by incorporating human knowledge into the pattern mining process.*

Limitation ②: The localization model is trained on source code as natural language token sequences, ignoring semantic information that is essential for accurately identifying where vulnerabilities may be injected in a given program. While the Transformer-based localization model in VulGen has shown to be reasonably capable of (e.g., faulty) code localization thanks to its explicit learning about locations through position encoding, the goal of effectively contextualizing vulnerability-injection code edits is hard to achieve with a vanilla Transformer position encoder. The reason is that code vulnerabilities are context-sensitive: the same lines of code (e.g., copying an array to a buffer) may cause a vulnerability when placed in one code context (e.g., where there is no boundary check against the buffer's size) but would not lead to any vulnerable behavior in another context (e.g., where the buffer has been ensured to be large enough to hold the array). For example, in Figure 1, the sample cannot be injected a "buffer-overflow" vulnerability by just removing Lines 7-8 if `password[BUFSIZE-1]` has been set to '\0'. Thus, whether pattern-based injection is fruitful depends on whether the pattern is applied in the right code context. *Identifying the right injection contexts is clearly reliant on code semantics, which can be learned by making the position encoding explicitly aware of semantic code information like value flows* [33] (e.g., value flow of `password` as illustrated in Figure 1).

3 TECHNICAL DESIGN

This section describes the technical design of VGX. We start with an overview of VGX and then describe the details of each module.

3.1 Overview

The overarching design goal of VGX is two-fold: (1) *large-scale*—being able to generate a massive number of samples to meet the thirsty need of training powerful defensive models, which requires the ability to inject vulnerabilities to normal code in the wild when following the injection-based methodology, and (2) *high-quality*—not including too many noisy samples (i.e., those that are considered vulnerable but actually not) among the generated ones, without which the large scale would not be meaningful. With respect to the advances made by prior works and the challenges they face (§2), VGX achieves this goal with the design shown in Figure 2. VGX consists of two main phases: *learning/training* and *production*.

In the *learning/training* phase, VGX gets itself built up through two main tasks: (1) train a deep learning based model that can identify the code contexts in a given (normal) program where vulnerabilities can be injected and (2) mine vulnerability-introducing code edit patterns that can materialize the injection in the identified contexts. These two tasks are achieved in two steps: *semantics-aware contextualization* and *human-knowledge-enhanced edit pattern formation*, followed by *preprocessing* the datasets needed by this phase.

In the *production* phase, with the edit patterns and trained contextualization model, VGX starts with *preprocessing* (using the same module) a given set of normal programs as input. Then, the model predicts potential vulnerability-injection code edit contexts, followed by matching and applying the most suitable edit patterns (for the contexts), resulting in (expectedly) vulnerable samples.

3.2 Preprocessing

To enable *semantics-aware contextualization* via the *customized Transformer*, we use *syntactic information* instead of processing code as natural language. Thus, we start by converting source code into Abstract Syntax Trees (ASTs), using tree-sitter [47] as the AST parser. An AST is a tree composed of nodes, each of which has a node type, a set of child nodes, and a value if it is a terminal.

Since Transformer takes text as inputs, we linearize the ASTs into text. As in [51], we do a pre-order traversal on the AST to get a sequence of node types and only keep the nodes above the expression level to reduce sequence length. We then concatenate the source code and the linearized AST for each sample with a special token `[SEP]` between them. We use this mixed/dual input as it helps learn the syntactic and contextual structures of code [51].

To make the *customized Transformer* use *semantic information*, which can be achieved via position encoding that is explicitly aware of value flow as discussed in §2, we construct a value flow graph (VFG) for each sample as in [33]—because it can capture vulnerability related code semantics [33, 71]. A VFG is a multi-edge graph

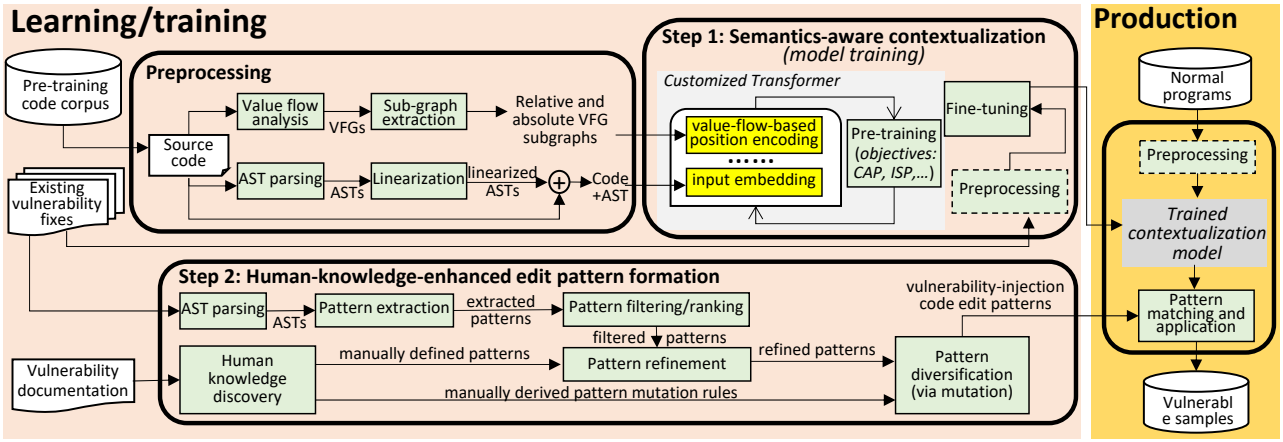

Figure 2: Design overview of the VGX approach, highlighting its two main phases: *learning/training* and *production*.

$g(V, E)$, where V is a set of nodes each representing a variable and E is a set of edges that each represent the value flows between two variables. For instance, in an assignment statement `c=a+b`, edges would exist from `a` to `c` and from `b` to `c`. The arrowed lines in Figure 1 show parts of the value flows in the motivating example.

Our value-flow-based position encoding is based on the traditional one, where absolute position encoding is calculated as per the absolute position (i.e., index) of a given token. Yet we calculate this encoding of a variable based on its respective *absolute VFG sub-graph*. To obtain this sub-graph, we traverse the VFG from the node of that variable until no new nodes can be found. For instance, in Figure 1, the token `password` at Line 9 has the value flow marked as red, and thus its absolute VFG sub-graph consists of itself and the node of token `password` at Line 1.

Similarly, we calculates the relative position encoding for a pair of variables ($v1$,$v2$) based on their respective *relative VFG sub-graph*. To obtain this sub-graph, we traverse the VFG from the node of $v1$ until we reach the node of $v2$. For example, in Figure 1, the pair of variables `buf` at Line 5 and `BUFSIZE` at Line 3 have their relative VFG sub-graph consist of the nodes of tokens `buf` at Line 5, `buf` at Line 3, and `BUFSIZE` at Line 3.

Next, the resulting absolute and relative VFG sub-graphs will be used for position encoding as elaborated in §3.3.1.

3.3 Semantics-Aware Contextualization

In this step, we train a `customized Transformer` in order to achieve *semantics-aware contextualization*. Given a normal program with the *Code+AST* input, we expect the model to output a code fragment that can be manipulated to introduce a vulnerability, based on the semantics and context of the code. To that end, the customized Transformer leverages *value-flow-based position encoding*, *pre-training*, and *fine-tuning* with data augmentation. We separately describe these in the following subsections.

3.3.1 Value-flow-based Position Encoding. In the encoder of our `customized Transformer`, the core module is self-attention blocks with position encoding:

$$z_i = \sum_{j=1}^{n} \frac{exp(a_{ij})}{\sum_{j'=1}^{n} exp(a_{ij'})} (x_j W^V + r_{ij}^V + r_{ij}^{V_{VFG}}) \quad (1)$$

$$a_{ij} = \frac{1}{\sqrt{2d}} (x_i W^Q)(x_j W^K + r_{ij}^K + r_{ij}^{K_{VFG}})^T + \frac{1}{\sqrt{2d}} (a_i^Q)(a_j^K)^T + \frac{1}{\sqrt{2d}} (a_i^{Q_{VFG}})(a_j^{K_{VFG}})^T \quad (2)$$

where z_i is the output hidden representation for the i-th token, a_{ij} is the attention between the i-th and j-th tokens; x_i and x_j are the hidden representation of the i-th and j-th tokens from the previous layer, respectively; W^V, W^Q, and W^K are the weight matrix of *value*, *query*, and *key*, respectively; r_{ij}^V, r_{ij}^K are the traditional relative position encodings and a_i^Q and a_j^K are the traditional absolute position encodings; and d is the dimension of the hidden representations. This traditional position encoding is solely based on the positions (indexes) of tokens hence *lacks semantic understanding of code*. For example, a variable is defined at one line but used many lines after. In this case, the distance $(i - j)$ used by the traditional relative position encoding may be too large, although the two variable tokens may share a direct definition-use relationship and have a close semantic (i.e., value-flow) distance.

To address this limitation, we utilize the VFG sub-graphs obtained during preprocessing to *incorporate semantic information into the position encoding*. Specifically, we add our value-flow-based relative position encoding $r_{ij}^{V_{VFG}}$ and $r_{ij}^{K_{VFG}}$, as well as value-flow-based absolute position encoding $a_i^{Q_{VFG}}$, and $a_j^{K_{VFG}}$ to the self-attention block, where each of the position encoding is the graph encoding of the respective sub-graph.

To compute the encoding of a given VFG sub-graph $g(V, E)$, we first use a pre-trained FastText [17] model for C language to convert the variable name in each node into an embedding. Then, we use a gated graph neural network (GGNN) to perform message passing aggregation and update the node embedding as follows:

$$x_v' = GRU(x_v, \sum_{(u,v) \in E} g(x_u)) \quad (3)$$

where GRU is the gated recurrent function [21], $g(.)$ is the function that assimilates the neighbor nodes' embedding, and x_u is the neighbor node of x_v. Then, we sum up embeddings of the nodes in the graph to get the VFG sub-graph embedding. Finally, the graph

embedding is multiplied with a weight matrix to get the *value-flow-based position encoding* $r_{ij}^{V_{VFG}}$, $r_{ij}^{K_{VFG}}$, $a_i^{Q_{VFG}}$, and $a_j^{K_{VFG}}$

Note that not all code tokens are variables. Thus, we only compute the value-flow-based position encoding when the token or both the tokens in the pair are variables. Otherwise, the value-flow-based position encoding is zero.

3.3.2 Pre-Training. To gain the awareness of code semantics, we performed pre-training before fine-tuning the model for *semantics-aware contextualization*. We use three existing general programming language-oriented objectives from CodeT5 [63] and introduce two new code contextualization-specific objectives (`CAP` and `ISP`).

CodeT5 Objectives. We utilize three pre-training objectives from CodeT5 [63] to learn general code comprehension for code-to-code transformation. The first is *mask span prediction* (`MSP`), where we randomly mask 15% of the source code tokens in an input that contains both source code and AST, with each mask having a span length uniformly ranging from 1 to 5 tokens. We train the model to recover the masked tokens based on the context. The second objective is *identifier tagging* (`IT`), where the model is trained to predict whether each source code token is an identifier. The third objective is *masked identifier prediction* (`MIP`), where we mask all the identifiers in the source code, and the model is trained to recover the identifiers based on the code semantics. We follow the pre-training approach of CodeT5, and feed the three pre-training objectives alternatively during training with an equal probability.

Code-AST Prediction (CAP). Since our model takes source code and ASTs as input, similar to SPT-Code [51], we adopt their approach and pre-train our model with the Code-AST Prediction (CAP) objective. We assign the correct AST to the corresponding source code in 50% of the pre-training samples, while in the other 50% we randomly assign an incorrect AST. Our model is trained to predict if the assigned AST corresponds to the input source code.

Irrelevant Statement Prediction (ISP). For each pre-training sample, we insert at a random location a randomly chosen statement from another sample. In this case, the functionality of the sample is not impacted by removing the inserted statement. Thus, the inserted statement is *semantically irrelevant* to the original sample. We pre-train the model to identify and output the irrelevant statement in each sample. We adopt this pre-training objective because it resembles our vulnerability injection localization objective. Since *vulnerabilities are context-sensitive based on code semantics, learning to differentiate (semantically) irrelevant statements (from relevant ones) intuitively helps identify the right code context that is (semantically) relevant to the vulnerabilities to be injected.*

We conduct pre-training in the following order: first against CAP, then the three CodeT5 objectives, and finally ISP, as justified by the inter-dependencies among these objectives. Also, for CAP and IT, the pre-training is done on the encoder only, while for all the other objectives (ISP, MSP, MIP) we pre-train both the encoder and decoder. These decisions are justified by the nature of the Transformer architecture.

3.3.3 Fine-Tuning. After pre-training, we fine-tune the *semantics-aware contextualization* model to locate code fragments that can be edited to introduce vulnerabilities, using the vulnerability-introducing code locations retrieved from the existing vulnerability fixes. To address the lack of fine-tuning data, we perform *data augmentation* through code refactoring on the fine-tuning samples. Following the approach in [54], we apply three types of *semantics-preserving refactoring*: (1) reverse the condition in an `if`-statement and swap the code in the `if` and `else` blocks; (2) convert a `for` loop into an equivalent `while` loop; and (3) insert unrelated junk code generated by SaBabi [61] to the code. We apply these transformations combinatorially on each original sample, leading to substantially more fine-tuning samples.

Table 1: Manually Defined Vulnerability-Injection Patterns

Patterns: *mutex*(h0); => EMPTY **Justification:** "Race Condition" mostly happens with a lack of mutex related statements, but there are many mutex related function [6]. Thus, once the located statement involve "mutex", we delete it.
Patterns: *TCHECK*(h0); => EMPTY *assert*(h0); => EMPTY **Justification:** There are many samples in the training set deleting statements involving "TCHECK" and "assert", but they usually use different function names[3]. Thus, once located statement involve "TCHECK" or "assert", we delete it.
Patterns: *free*(h0); => EMPTY *Free*(h0); => EMPTY *destruct*(h0); => EMPTY *destroy*(h0); => EMPTY *unref*(h0); => EMPTY *clear*(h0); => EMPTY **Justification:** "Memory Leak" mostly happens with not releasing assigned memory. However, there may be many different functions for releasing the memory [4]. Thus, once the the located statement involve memory release related functions, we delete it.
Patterns: unsigned h0; => h0; int64_t h0; => int h0; static h0 h1 = h2; => h0 h1 = h2; **Justification:** "Type Error" usually happens with not using static, unsigned, large-size types, but the current patterns specify too many details like identifier names or assigned values in the patterns [7]. Thus, we make these details holes so that they are more general.
Patterns: memset(h0); => EMPTY h0 = *ERR*; => EMPTY h0 = *NONE* => EMPTY h0 = 0; => EMPTY h0 = NULL; => EMPTY *buf* = h0; => EMPTY **Justification:** "Use of Uninitialized Variables" usually happens with not initializing declared variables, but current patterns specify too many details like identifier names and values in the patterns [8]. Thus, we make these details holes and use regular expression to represent the common initialized value, so that they are more general.
Patterns: h0 = kcalloc(hole1, hole2, hole3); => h0 = kzalloc(h1*h2, h3); h0 = calloc(hole0, hole1); => h0 = malloc(h1*h2); **Justification:** "Memory Allocation Vulnerability" usually happens when using unsafe memory allocation functions, but current patterns specify too many details like identifier names and values in the patterns [5]. Thus, we make them holes to make the patterns more general.

3.4 Edit Pattern Formation

This section describes the process of VGX on learning (from the existing vulnerability fixes) to materialize the code editing for realizing vulnerability injection, including *pattern extraction*, *pattern filtering/ranking*, and *pattern refinement and diversification.*

3.4.1 Pattern Extraction. We first parse the source code into ASTs like the one for *semantics-aware contextualization* but with a different AST parser srcML [22], as it better supports edit pattern extraction and application [14, 55]. Then, we follow the approach in Getafix [14] to extract the edit patterns via anti-unification. Because of the space limit, we refer readers to the original Getafix paper [14] for details. An edit pattern is a pair of code fragments in terms of ASTs representing a code edit. For example, "$h0[h1-1] = 0; \Longrightarrow h0[h1] = 0;$" represents removing "`-1`" from the code where `h0` and `h1` are placeholders which can match any identifiers and literals. After the pattern extraction, we get many edit patterns that range from very general (i.e., the patterns that can match and apply on many different code samples) to very specific (i.e., the patterns that can only match and apply on a few code samples) [14].

3.4.2 Pattern Filtering/Ranking. To obtain the appropriate edit patterns for introducing vulnerabilities, we establish rules to filter and rank the extracted patterns. First, we compute three scores for each of the extracted edit patterns:

(1) **Prevalence score (s_{preval})**: Assuming knowing the vulnerabi-lity-introducing locations, the prevalence score is the

number of samples that can be injected the vulnerabilities correctly by this pattern in the training set, as it is proportional to the probability that the pattern can inject vulnerabilities successfully.

(2) **Specialization score (s_{spec})**: In the training samples, we compute the average number of AST subtrees that the pattern can match. The reciprocal of the average number is the specialization score, as it indicates whether the pattern is specific so that it will not match AST subtrees which cannot be injected vulnerabilities.

(3) **Identifier score (s_{ident})**: The identifier score is the number of specified identifier names in the pattern, as the identifier names inform about the code semantics significantly, which are crucial for correctly injecting vulnerabilities.

The product of the three scores is the final ranking score.

$$s_{rank} = s_{preval} \times s_{spec} \times s_{ident} \quad (4)$$

Then, we rank the edit patterns by s_{rank} in a non-ascending order. The three individual scores are not normalized since only relative rankings matter. The higher the score, the more likely the pattern can successfully inject vulnerabilities. Finally, we only keep the top 300 edit patterns, so as to filter out those that are not likely to inject vulnerabilities successfully.

3.4.3 Pattern Refinement and Diversification. While the patterns extracted and filtered/ranked enable to inject vulnerabilities in some cases, they do not fully capture the knowledge about how real-world vulnerabilities can be introduced to software as embodied in the extant vulnerability documentation such as CWE/CVEs in NVD and issues/bug reports on GitHub, because the available datasets of vulnerability-introducing edits are still limited, as discussed in §2. To address this limitation, we refine the patterns. We applied the 300 patterns to the vulnerability-introducing training samples, assuming knowing the vulnerability-introducing locations. As a result, we encountered *false positives*, where the pattern was applied but the generated sample was not actually vulnerable, as well as *false negatives*, where none of the 300 patterns could be applied.

For the false positives, we consider that the patterns applied on them are too general. We removed the patterns against which more than half of the applications are false positives. In total, we removed 21 patterns in this manner.

For the false negatives, we consider that the corresponding patterns are too specific. Thus, we manually define new patterns that are necessarily more general. Specifically, we examined the vulnerability-fixing commit associated with the false-negative training sample and identified the corresponding CWE ID [2]. Then, we read the CWE documentation and check the examples used to describe that CWE. We further examined other real-world vulnerability samples with the same CWE ID in the NVD/CVE database [50]. Based on the false negative, the examples, and other real-world samples, we compose several possible patterns that would be able to inject the respective vulnerabilities. Then, we went back to the existing pattern set and found patterns close to the composed ones. We finally modified those patterns into the composed ones with the use of regular expression (regex). These modified patterns are the 20 manually defined, new patterns described in Table 1.

Since manual refinement is costly, we are not able to derive many new patterns. Some remaining patterns are still too specific in syntactic structure. Thus, we further derived a set of pattern mutation rules to make the edit patterns sufficiently general. We carefully checked the traditional mutation operators for C language [13]. We then selected those that do not change the code functionality significantly and finally derived four types of pattern mutation rules. Table 2 shows the pattern mutation rules we derived and the respective justifications for them. The red ⇔ means that the mutation is bidirectional, while ⇒ means that the mutation is unidirectional. After the mutation, we obtained a total of 604 high-quality *vulnerability-injection patterns*.

Table 2: Pattern Mutation Rules

Rule: function_name(parameters); =>new_function_name(new_parameters); ⇔ h0=function_name(parameters); => h0=new_function_name(new_parameters); **Justification:** Some function calls like strncpy may return some values but the return values do not always assign to a variable. Mutating such patterns so that they have or remove the return value assignments increases the generalizability of the pattern set.
Rule: if(condition) return NULL/0/-1; => EMPTY ⇔ if(condition) return -EINVAL/EBADFD/ENOTSOCK/EPERM/ENODEV/ENOMEM; => EMPTY **Justification:** The typical safety issue checks do some check in an if statement condition and then return an error code when found the issue. However, there are many possible returned error codes. We mutate these returned error codes to make the patterns more general.
Rule: if(specific_condition) return error_code; => EMPTY ⇒ if(hole) return error_code;=> EMPTY **Justification:** In our automatic pattern mining, the mined safety checks if statements may be too specific in the condition. However, if an if statement only has one return statement, it is very likely that it is a safety check if statement. Thus, we remove the specific conditions in these if statements and use a hole to make the patterns more general.
Rule: if(condition) return error_code; => EMPTY ⇔ if(condition) break/continue; => EMPTY **Justification:** When the safety checks if statements find issues, they may not always exit the function using a return. If the safety check is in a for/while/switch block, it may use a break or continue to exit. Thus, we mutate the exit statement to make the patterns more general.

Currently, the pattern refinement and pattern mutation-rule derivation are carried out manually and aimed at the C language. Yet the process can be readily reproduced and replicated by following the principled steps laid out above. This also largely makes our VGX approach adaptable for other programming languages.

3.5 Vulnerability Production

With the trained *contextualization* model and the *vulnerability-injection code edit patterns*, VGX is ready to produce vulnerability data. Given a normal program, we again preprocess it like we do in the learning/training phase. Next, the *contextualization* model identifies a code fragment (as the context) for vulnerability injection. Then, VGX applies the most suitable pattern to the identified context. To do so, we rank the patterns in the final set of *vulnerability-injection code edit patterns* by the score computed in Equation 4 and apply the first pattern in the set that can match the context. Note that VGX only injects the vulnerability when one of the patterns matches the context. If none of the patterns matches, VGX discards the sample to reduce false positives. Otherwise, the program with the pattern applied is expected to be vulnerable.

4 EVALUATION

We evaluate the effectiveness of VGX for generating vulnerability data. We seek to answer the following research questions:

- **RQ1:** How effective is VGX in vulnerability generation compared to other approaches?
- **RQ2:** How do the novel design components contribute?
- **RQ3:** How efficient is VGX in vulnerability generation?

4.1 Implementation

To ensure the accuracy/reliability of the value flow graphs created in the preprocessing module, we leverage the value flow extraction tool from GraphCodeBERT [33] to construct the value flow graph

of each sample. We modify the source code of TPTrans [58] to build our *customized Transformer* as it provides position encoding implementation which is easy to adapt/customize. We use the Getafix implementation in VulGen [55] for the anti-unification-based pattern extraction. Our experiments were performed on a machine with a 32 Cores AMD Ryzen 3970X (3.7GHz) CPU, two Nvidia RTX 3090 GPUs, and 256GB DDR memory.

4.2 Dataset

For the *pre-training* in *Semantics-Aware Contextualization*, we use the IBM CodeNet [59] dataset. We extract 1,213,907 functions in C language from it to build the pre-training dataset. For the *fine-tuning* in *Semantics-Aware Contextualization* and *Edit Pattern Formation*, we use the dataset in VulGen [55] which is the combination of five widely used vulnerability datasets. We remove the overlapped samples and eventually obtain 7,764 samples for the evaluation. We split the dataset by 9:1 for training (6,989) and testing (775). We then augment (§3.3.3) the 6,989 samples into 156,665 for *fine-tuning*.

4.3 Metrics

Given that a high-quality vulnerability sample should be a truly exploitable program [23], we evaluate whether the code changes make the program exploitable for the attackers. For the samples exactly-matching their ground truths, they are known as exploitable because they are confirmed real-world vulnerabilities. For these cases, we evaluate vulnerability generation in terms of precision, recall, and F1. We compute: $precision = \frac{\#matched\ samples}{\#generated\ samples}$; $recall = \frac{\#matched\ samples}{\#testing\ samples}$ and $F1 = \frac{2 \times recall \times precision}{precision + recall}$.

Since the generated samples may be exploitable although do not exactly match the ground truth [54, 55], we randomly sampled and manually inspected some of them to compute the *success rate* $= \frac{\#exploitable\ samples}{\#generated\ samples}$ where the *exploitable samples* also include the *matched samples*. We inspected 258 cases, a sample size that is statistically significant at 95% confidence level and 5% margin of error with respect to the population (i.e., testing set) size of 775.

The manual inspection of a sample assesses if an exploit can be written to attack it. Yet manually writing exploits for all the non-exactly-matching samples generated by VGX and baseline techniques is quite difficult. Thus, we wrote exploits for 30% of those samples in RQ1 and RQ4. Each exploit is an executable that can attack the generated program but not the normal one, ensuring the exploitability is introduced by the code change. This process helps us learn how to decide whether a sample is exploitable, based on which we determined the exploitability of the remaining 70% without actually writing/running the exploits. The inspection was first done by the first author, checked by the third and fourth authors, and then verified/calibrated by all three to ensure the correctness.

4.4 RQ1: Effectiveness of VGX

We assess the effectiveness of VGX over four baselines:

- **VulGen** [55] is a vulnerability generation tool which fine-tunes CodeT5 [63] for localization and uses Getafix [14] to mine edit patterns for vulnerability injection.
- **CodeT5** [63] is a pre-trained model good at code-to-code transformation, which can be directly fine-tuned to generate vulnerable code for given normal programs.
- **Getafix** [14] is a pattern mining and application technique for bug fixing. We directly use it for vulnerability generation.
- **Graph2Edit** [65] is a code editing model taking a program's AST as input and predicting a sequence of edits on it, which can be trained for vulnerability injection.

Table 3: Effectiveness (and improvements in parentheses) of VGX over the baselines for vulnerability generation in RQ1.

Technique	Precision	Recall	F1	Success Rate
VGX	59.46%	22.71%	**32.87%**	**93.02%**
VulGen	17.50% (239.77%↑)	15.74% (44.28%↑)	**16.51% (99.09%↑)**	**75.96% (22.45%↑)**
CodeT5	12.65% (370.04%↑)	12.65% (79.53%↑)	**12.65% (159.84%↑)**	**24.81% (274.93%↑)**
Getafix	4.67% (1173.23%↑)	2.58% (780.23%↑)	**3.32% (890.06%↑)**	**57.75% (67.07%↑)**
Graph2Edit	13.97% (325.62%↑)	13.97% (65.56%↑)	**13.97% (135.29%↑)**	**21.71% (328.47%↑)**

Table 3 shows the effectiveness of VGX and the baselines against the 775 testing samples. VGX generates 296 samples, of which 176 exactly match the ground truth. Thus, the precision, recall, and F1 is 59.46%, 22.71%, and 32.87%, respectively. The high precision indicates the good quality of the generated samples, and the 22.71% recall suggests the generalizability for supporting large-scale vulnerability generation. The 32.87% F1 shows the overall promising effectiveness of VGX—99% higher than the best baseline. Notably, Column 3 shows VGX's success rate of 93.02%, indicating that the vast majority of its generated samples are vulnerable/exploitable.

Table 3 Rows 3-6 show the effectiveness of the four baselines, and the numbers in parentheses indicate VGX's relative improvements over the baselines. VGX outperforms the GNN-based code editor Graph2Edit by substantial margins. While Graph2Edit is a general-purpose code editor with a promising editing process (i.e., predicting a sequence edits on the AST) [54], it suffers from the lack of training samples as GNN needs a large number of samples to be reasonably trained [55]. VGX outperforms the CodeT5 and Getafix also quite significantly. This may be justified by the merits of combining deep learning to locate injection with a pattern-based method to materialize injection edits. While VulGen also works in an overall similar way, VGX still greatly outperforms it. This is due to VGX taking the *advantages of human-knowledge-enhanced edit patterns and semantics-aware contextualization, which largely overcome VulGen's Limitation ① and Limitation ②, respectively.*

4.5 RQ2: Contributions of Novel Components

In this section, we investigate the contribution of each novel component in VGX through ablation studies. We remove each of those components and compare the effectiveness before and after the removal. The results are shown in Table 4. To show the impacts of the contextualization designs, we also report the localization accuracy in Column 2. The numbers in parentheses indicate VGX's relative improvements compared to the ablated versions.

To assess the impact of the *contextualization-specific pre-training objectives*, we remove the pre-training for `CAP` and `ISP`. Table 4 Row 3 shows the results. With the two pre-training tasks, VGX improves the localization accuracy by 4.37%, and thus improves the vulnerability generation by 9.38%, 8.55%, and 8.87% in terms

Table 4: The contribution of VGX's components for vulnerability generation. The numbers in parentheses are VGX's relative improvements compared to the ablated versions.

Experiment	Loc Acc	Precision	Recall	F1
VGX	55.35%	59.46%	22.71%	**32.87%**
No CAP and ISP	53.03% (4.37%↑)	54.36% (9.38%↑)	20.90% (8.66%↑)	**30.19% (8.87%↑)**
No AST	49.67% (11.44%↑)	53.90% (10.32%↑)	19.61% (15.81%↑)	**28.76% (14.29%↑)**
No VFG	52.13% (6.18%↑)	53.29% (11.58%↑)	21.93% (3.56%↑)	**31.07% (5.79%↑)**
No Augmentation	51.61% (7.25%↑)	53.33% (11.49%↑)	19.61% (15.81%↑)	**28.67% (14.65%↑)**
No Diversification	55.35% (0.00%↑)	62.45% (-4.78%↑)	20.38% (11.43%↑)	**30.73% (6.96%↑)**
No Refinement	55.35% (0.00%↑)	71.91% (-17.31%↑)	16.51% (37.55%↑)	**26.85% (22.42%↑)**

of precision, recall, and F1, respectively. This indicates that the contextualization-specific objectives are effective for pre-training.

To understand the contribution of the *linearized AST*, we remove the AST part in the Transformer input and only use the source code. Table 4 Row 4 shows the results. With linearized AST, VGX improves the localization accuracy by 11.44%, and thus improves the vulnerability generation effectiveness by 10.32%, 15.81%, and 14.29% in terms of precision, recall, and F1, respectively. This indicates that adding the linearized AST to the input text effectively helps the model understand the syntactic structure of the code, hence notably improving vulnerability injection.

To show the effectiveness of the *VFG-based position encoding*, we remove it and only use traditional position encoding in the Transformer model. Table 4 Row 5 shows the results. With our new position encoding, VGX improves the localization accuracy by 6.18%, and thus improves the vulnerability generation effectiveness by 11.58%, 3.56%, and 5.79% in terms of precision, recall, and F1, respectively. This indicates that the VFG-based position encoding helps the Transformer understand the semantics of the code, thus improves the vulnerability generation significantly.

To measure the impact of the *data augmentation when fine-tuning* the contextualization model, we use the original 6,989 training samples only to train it. Table 4 Row 6 shows the results. With the data augmentation, The localization accuracy improves by 7.25%, and thus improves the vulnerability generation effectiveness by 11.49%, 15.81%, and 14.65% in terms of precision, recall, and F1, respectively. This indicates that the data augmentation mitigates the lack of training data, thus is helpful for vulnerability generation.

To see the impact of *pattern diversification*, we only use the edit patterns prior to the diversification (mutation). Table 4 Row 7 shows the results. While pattern diversification makes the precision decrease by 4.78%, the recall and F1 improve by 11.43% and 6.96%, respectively. While the mutation makes the vulnerability injection more general, it also makes the patterns less restrictive against the injection context identified. The improvement of recall indicates that VGX is able to generate more vulnerable samples, which is important for generating large-scale vulnerability datasets.

We further rollback the edit pattern set to the one without *pattern refinement*. Table 4 Row 8 shows the results. We can see the recall and F1 further decrease to 16.51%, and 26.85%, respectively, while the precision increases to 71.91%. This indicates that patterns without refinement are too strict and specific, making it difficult for VGX to generate large-scale vulnerability data. Thus, our refinement improves the generalizability of the patterns, making the overall vulnerability generation more scalable.

4.6 RQ3: Efficiency of VGX

As VGX, VulGen, and Getafix support multiprocessing generation but CodeT5 and Graph2Edit do not, we use single-process generation for a fair comparison. On average, VGX injects vulnerabilities on 189.02 samples per minute, while the numbers for VulGen, Getafix CodeT5, and Graph2Edit are 132.85, 276.78, 29.88, and 89.08, indicating VGX is comparable to the baselines in terms of efficiency.

5 USEFULNESS OF VGX

We use VGX to generate a large-scale vulnerability dataset and evaluate the usefulness of the data for training downstream DL-based vulnerability analysis tools. As VulGen is also designed for vulnerability data generation and is shown to be relatively effective in the original paper [55], we also directly compare VGX's usefulness with VulGen. We answer the following research questions:

- **RQ4**: How effective is VGX in generating large-scale vulnerable data using normal programs in the wild?
- **RQ5**: Can the generated samples improve the downstream DL-based vulnerability analysis tools?
- **RQ6**: Can the improved vulnerability detection models find more latest real-world vulnerabilities?

5.1 RQ4: Large-Scale Production

To get a large number of normal samples for VGX to generate vulnerability data, we follow the approach in [26] to collect the latest source code of 238 projects that are involved in the CVE/NVD database [50]. Then, we extract the functions in these projects and obtain 738,453 normal functions for vulnerability injection.

We apply VGX on these functions and it generates **150,392 samples** in 50 hours 48 minutes. We randomly sample some to manually inspect. With the same procedure in §4.3, the sample size is 375. Again, we write exploits for 30% (102) of the samples and then directly label the remaining samples. The success rate is **90.13%**, which is close to the success rate (93.03%) for RQ1 (§4.4). In comparison, VulGen generates **686,513 samples**. With the sample size 384, the success rate is **41.33%**. This is not only much lower than VGX's, but also much lower than its own (75.96%) in §4.4. The reason is that VulGen's edit patterns are vague hence applied on the statements that cannot be injected vulnerabilities (Limitation ①) and the localization model is not semantics-aware hence less accurate (Limitation ②). This shows that VGX is effective to generate large-scale vulnerability data from the wild while VulGen is not.

We also checked the vulnerability types represented by these 375 samples and found that they cover 23 types (CWEs). Figure 3 shows the distribution of them. It is worth noting that these types include many of the most (e.g., top-25) dangerous ones according to the recent CWE report [2], such as CWE-787, CWE-20, CWE-125, etc. This indicates that our generated vulnerability samples are diverse.

5.2 RQ5: Downstream Analysis Improvement

Given the vulnerability generation process, our data supports model training for at least three downstream tasks: (1) *function-level vulnerability detection*; (2) *line-level vulnerability localization*; and (3) *vulnerability repair*. We use the generated data to augment the training sets of the models and show the improvement after the augmentation. Since some models are designed for the existing

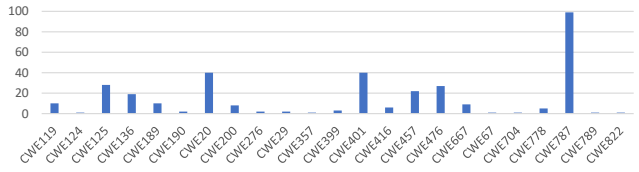

Figure 3: The vulnerability type (CWE) distribution of generated vulnerable samples in the large-scale production.

Table 5: Improvement of vulnerability detection using VGX's, VulGen's generated samples, and samples from SARD.

Model	Precision	Recall	F1
Devign-ori	9.82%	50.19%	**16.43%**
Devign-aug-VGX	12.37% (25.97%↑)	52.47% (4.54%↑)	**20.01% (21.79%↑)**
Devign-aug-VulGen	11.23% (14.35%↑)	30.03% (-40.17%↑)	**16.35% (-0.49%↑)**
Devign-aug-SARD	15.27% (55.49%↑)	15.21% (-69.69%↑)	**15.24% (-7.24%↑)**
LineVul-ori	26.42%	2.52%	**4.61%**
LineVul-aug-VGX	11.38% (-56.93%↑)	78.00% (2995%↑)	**19.86% (330.80%↑)**
LineVul-aug-VulGen	9.97% (-62.26%↑)	3.73% (48.01%↑)	**5.42% (17.57%↑)**
LineVul-aug-SARD	9.19% (-65.21%↑)	85.70% (3300%↑)	**16.60% (260.09%↑)**
IVDetect-ori	9.06%	75.52%	**16.18%**
IVDetect-aug-VGX	13.21% (45.81%↑)	35.66% (-52.78%↑)	**19.28% (19.15%↑)**
IVDetect-aug-VulGen	7.90% (-12.80%↑)	65.03% (-13.89%↑)	**14.09% (-12.92%↑)**
IVDetect-aug-SARD	10.04% (10.81%↑)	55.94% (-25.92%↑)	**17.02% (5.19%↑)**

small vulnerability datasets, they are not scalable for our large dataset in terms of the time cost and the memory usage. Thus, we use 10% of VGX's generated samples (15,039) and VulGen's generated samples (68,651) to improve the downstream models. To show that VGX's generated samples are more practical than the existing synthetic samples, we also use the same number (15,039) of samples from the widely used synthetic dataset SARD [1] to improve the downstream tasks for comparison. Since the original testing sets of these models are split from their training sets which have the same distribution, testing the models on them may not show their real performance [56]. Thus, we follow the approaches in [54, 55] to leverage the *independent testing* setting: for each downstream task, we use a third-party testing set that is from a different source from the original training sets. Again, we ensure that there is no overlapping between the training sets and testing sets.

5.2.1 Function-Level Vulnerability Detection. We improve three vulnerability detectors, Devign [71], LineVul [28], and IVDetect [43], as they are the SOTA tools publicly available for replication experiments. The original training set of each tool has vulnerable and non-vulnerable samples. To improve each training set, we add the 15,039 generated samples by VGX and the *same proportion* of non-vulnerable samples from our normal sample set used to generate vulnerable data. Specifically, the training set of Devign has 9,744 vulnerable and 11,012 non-vulnerable samples, thus we add VGX's 15,039 generated samples and 16,996 non-vulnerable samples to the training set. Both LineVul and IVDetect use the Fan [26] training set with 10,547 vulnerable and 168,752 non-vulnerable samples, thus we add VGX's 15,039 generated samples and 240,624 non-vulnerable samples to the training set to improve the models.

To leverage *independent testing*, we test the models on the ReVeal dataset [19]. It has 1,664 vulnerable and 16,505 non-vulnerable samples which are manually collected from real-world projects. Table 5 shows the results of the trained models of the three tools, where *-ori* means that the model is trained on the original training set and *-aug-VGX* means that the model is trained on the augmented training set using VGX's generated samples. We use the most widely adopted metrics for the vulnerability detection models, which are precision, recall, and F1, to assess the improvement. We notice that the F1 scores, which represent the overall performance of the tools, improve by 21.79%, 330.80%, and 19.15% for Devign, LineVul, and IVDetect, respectively. This indicates that VGX's generated data is able to improve vulnerability detection significantly.

We also do the same process with VulGen's generated samples and SARD samples. We notice that the improvements are much lower than the one with VGX's generated samples (e.g., 17.57% and 260.09% versus 330.80% F1 against LineVul) and they may even decrease the performance (e.g., 0.49% and 7.24% F1 decrease against Devign). Besides, VulGen's generated samples may be even worse than the SARD samples. The reason is that VulGen's generated samples have a low success rate (41.33%) for large-scale production and the label noise weakens the data augmentation seriously.

Table 6: Improvement of vulnerability localization using VGX's, VulGen's generated samples, and samples from SARD.

Model	Top-10 Accuracy	Model	Top-10 Accuracy
LineVul-ori	48.84%	**LineVD-ori**	59.25%
LineVul-aug-VGX	58.27% (19.31%↑)	**LineVD-aug-VGX**	66.87% (12.86%↑)
LineVul-aug-VulGen	53.43% (9.39%↑)	**LineVD-aug-VulGen**	52.68% (-11.09%↑)
LineVul-aug-SARD	49.85% (2.07%↑)	**LineVD-aug-SARD**	64.18% (8.32%↑)

Table 7: Improvement of vulnerability repair using VGX's, VulGen's generated samples, and samples from SARD.

Model	Top-1 Accuracy	Top-5 Accuracy	Top-50 Accuracy
VulRepair-ori	8.55%	11.81%	16.29%
VulRepair-aug-VGX	21.05% (146.20%↑)	29.12% (146.57%↑)	30.14% (85.02%↑)
VulRepair-aug-VulGen	11.81% (38.13%↑)	16.77% (41.20%↑)	17.85% (9.85%↑)
VulRepair-aug-SARD	11.07% (29.47%↑)	13.92% (17.87%↑)	17.18% (5.46%↑)
VRepair-ori	2.58%	5.16%	8.62%
VRepair-aug-VGX	4.41% (70.93%↑)	10.59% (105.23%↑)	17.18% (99.30%↑)
VRepair-aug-VulGen	2.85% (10.46%↑)	7.26% (40.70%↑)	14.05% (62.99%↑)
VRepair-aug-SARD	1.36% (-47.28%↑)	3.46% (-32.94%↑)	4.96% (-42.45%↑)

5.2.2 Line-Level Vulnerability Localization. We improve two vulnerability localization tools LineVul [28] and LineVD [35] with VGX's generated samples. Their training set is also Fan [26] mentioned in §5.2.1. Thus, we use the same augmentation setting. We again test the models on the ReVeal dataset because it provides the vulnerable lines for the 1,664 vulnerable samples. Table 6 shows the results of before and after the improvement. We adopt the commonly used metric in LineVul and LineVD, which is top-10 accuracy, to evaluate the improvement. The improvements brought by VGX's generated samples are 19.31% and 12.86% for LineVul and LineVD, respectively. This indicates that the samples are useful for improving vulnerability localization. In comparison, VulGen's generated samples and SARD samples have lower improvements compared to VGX's generated samples or even decrease performance (9.39% and 2.07% improvements for LineVul, respectively; and -11.09% and 8.32% improvements for LineVD, respectively).

5.2.3 Vulnerability Repair. We improve the latest two vulnerability repair tools VulRepair [29] and VRepair [20]. Both use the vulnerability repair samples from the combination of Fan [26] and CVEFixes [15], which consist of 8,482 pairs of vulnerable and the respective non-vulnerable programs, to train the models. We thus use

Table 8: Latest CVEs Detected by Improved LineVul but missed by the original one.

CVE ID	Project	CWE ID	CVE-ID	Project	CWE ID
2022-46149	Cap'n Proto	CWE-125	2021-3764	Linux Kernel	CWE-401
2023-27478	libmemcached-awesome	CWE-200	2022-47938	Linux Kernel	CWE-125
2022-28388	Linux Kernel	CWE-415	2023-23002	Linux Kernel	CWE-476
2023-22996	Linux Kernel	CWE-772	2022-42895	Linux Kernel	CWE-824
2021-3743	Linux Kernel	CWE-125	2022-34495	Linux Kernel	CWE-415
2022-24958	Linux Kernel	CWE-763	2022-47520	Linux Kernel	CWE-125
2022-30594	Linux Kernel	CWE-863			

the 15,039 pairs of VGX's generated vulnerable and the respective non-vulnerable programs to augment the training set. We test the models on the PatchDB dataset [62] because its vulnerability repair samples are from real-world projects and confirmed by humans. Table 7 shows improvement results. We adopt the commonly used settings where the beam search sizes are 1, 5, and 50 and thus they report top-1, top-5, top-50 accuracy. With VGX's generated samples, the top-1, top-5, and top-50 accuracy of VulRepair improves by 146.20%, 146.57%, and 85.02%, respectively. The top-1, top-5, and top-50 accuracy of VRepair improves by 70.93%, 105.23%, and 99.30%, respectively. This indicates that the samples are quite useful to boost vulnerability repair. In comparison, VulGen's generated samples and SARD samples bring lower improvements or even decrease the performance (e.g., 9.85% and 5.46% improvements for VulRepair's Top-50 accuracy, respectively; and 62.99% and -42.45% for VRepair's Top-50 accuracy, respectively).

5.3 RQ6: Real-World Vulnerability Discovery

We scraped the latest 71 vulnerabilities covering 17 CWEs from 6 critical software projects (e.g., Linux kernel) reported between 2021-2023, from the CVE/NVD database [50]. We use LineVul as it is the latest available vulnerability detector. Table 8 shows the latest vulnerabilities detected by the LineVul model improved by VGX's generated samples but cannot be detected by the original one. The improved LineVul found *13 more CVEs*, indicating its enhanced potential for discovering real-world zero-day vulnerabilities.

6 DISCUSSION

In this section, we discuss why VGX has good performance on generating vulnerability data and why the generated samples by VGX effectively improve the performance of downstream tasks.

6.1 Vulnerability Generation Performance

As discussed in §3, the key designs of VGX are *semantics-aware contextualization* and *human-knowledge-enhanced edit patterns formation*. We dissect VGX's performance merits from these aspects.

The value-flow-based position encoding makes the contextualization semantics-aware. Figure 4 shows an example where VGX correctly predicts the vulnerability injection location due to its value-flow-based position encoding. The reason is that this encoding explicitly enhances the attention between variables that have value flow relationships. For example, in the case shown in the figure, the variable `res` has value flows from `pdev`, `IORESOURCE_MEM`. These variables are highly related to vulnerability injection. Thus, while they are not in the statement to be located, VGX pays more attention to these variables when locating the correct statement. However, the one without value-flow-based position encoding simply locates Line 2 because it has the tokens `dev, pdev`,

```
1  static int hi3660_stub_clk_probe(struct platform_device *pdev){
2      struct device *dev = &pdev->dev;
3      struct resource *res;
4      stub_clk_chan.cl.dev = dev;
5      stub_clk_chan.mbox = mbox_request_channel(&stub_clk_chan.cl, 0);
6      if (IS_ERR(stub_clk_chan.mbox)) return PTR_ERR(stub_clk_chan.mbox);
7      res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
8      if (!res) return -EINVAL;
9      freq_reg = devm_ioremap(dev, res->start, resource_size(res));
10     if (!freq_reg) return -ENOMEM;   ...
11 }
```

Figure 4: An example where VGX correctly predicts the statement at Line 8 (marked as cyan), but without value-flow-based position encoding it would incorrectly locates Line 2 (marked as yellow).

```
1 static void __del_gref(struct gntalloc_gref *gref){ …
2     gref->notify.flags = 0;
3     if (gref->gref_id > 0){
4         if (gnttab_query_foreign_access(gref->gref_id)) return;
5         if (!gnttab_end_foreign_access_ref(gref->gref_id, 0)) return;
6         gnttab_free_grant_reference(gref->gref_id);  ...
7 }
```

Figure 5: An example where VGX successfully remove the statement at Line 6 (marked as cyan) to inject a memory leak vulnerability, but without manual pattern refinement via regex it would not be able to do so.

and `device`, which appeared on vulnerable statements of some training samples, but that statement has nothing to do with injecting a vulnerability as per the code semantics here. VGX takes the advantage of the value-flow-based position encoding which makes the model semantics-aware hence better contextualization performance, overcoming Limitation ② for vulnerability generation.

Manual pattern refinement with regex makes the patterns necessarily general hence more useful. Figure 5 shows an example where VGX successfully removes the statement at Line 6 (marked as cyan) to inject a "memory leak (CWE-401)" vulnerability but the one without manual pattern refinement via regex cannot. The reason is that this program uses a self-defined `free` function to release the pointer. Thus, the patterns mined through anti-unification may not cover the function name of this `free` function. The use of self-defined functions related to memory allocation, memory initialization, memory release, safety check, and multi-threading management are common based on our manual inspection. Thus, our manually added edit patterns with regex shown in Table 1 make the edit patterns necessarily more general, improving the performance of VGX for vulnerability generation, which effectively overcomes Limitation ① for vulnerability generation.

Pattern diversification makes the patterns more diverse hence more applicable. Figure 6 shows an example where VGX successfully injects a CWE-20 vulnerability due to its pattern diversification. The reason is that the input validation checking at Line 4 may be very different in different programs. It is difficult for the patterns extracted from existing examples to cover all of them. Yet based on our *human knowledge*, the return value `-EINVAL` has indicated that this is a value validation checking. Thus, we build pattern mutation rules (Table 2 Row 4) such that once such return error values are in the `if` statement, we do not need to match the condition any more. Therefore, the pattern diversification integrates our human knowledge, making the edit patterns

```
1 int vrend_create_vertex_elements_state(…,unsigned num_elements,…){
2     struct vrend_vertex_element_array *v = CALLOC_STRUCT(…);
3     if (!v)  return ENOMEM;
4     if (num_elements > PIPE_MAX_ATTRIBS)  return -EINVAL;
5     v->count = num_elements;
6     ...
7 }
```

Figure 6: An example where VGX successfully remove the if statement at Line 4 (marked as cyan) to inject an improper input validation (CWE-20) vulnerability, but without pattern mutation it would not be able to inject correctly.

more diverse and applicable hence improving VGX for vulnerability generation, overcoming Limitation ① of peer approaches.

6.2 Usefulness of Generated Samples

In §5, we show that VGX-generated samples are high-quality and thus are effective for improving various downstream vulnerability analysis tasks. Specifically, our generated samples have merits in four main aspects of high dataset quality:

Dataset size. Training dataset size is crucial for training a deep learning model well. Almost all the DL-based code analysis techniques use more than 10,000 samples to train the models [16]. However, previous works [54, 55] like VulGen can only generate a small number of vulnerability samples (e.g., <1,000) or their quality plummets. In contrast, VGX can generate large-scale quality vulnerability samples in a short time. The large number of generated samples allows the DL model to learn relevant knowledge in general.

Complexity. Some previous vulnerability analyses use synthetic datasets to train their models [45, 46]. However, synthetic samples are usually very simple and make the trained models not generalizable to real-world vulnerability analysis [19]. This is also confirmed in our experiments in §5.2. In contrast, VGX's generated samples are based on real-world normal samples, thus the generated samples are as complex as real-world vulnerability data and ensure the model learns the knowledge for challenging vulnerability analysis.

Noise. Another important aspect of the usefulness is noise. While VulGen [55] can generate vulnerable samples, the success rate is low (41.33%), making the samples unready to use. In contrast, VGX's generated samples achieve a 90.13% success rate, which can be used to improve the model training of downstream tasks directly.

Diversity. Previous works [67, 68] only generate one or a few number types of vulnerabilities, which cannot train vulnerability analysis models in general. In contrast, VGX is already able to generate vulnerability samples of diverse types (CWEs) while spanning a variety of (238) projects, making its augmented model training for downstream vulnerability analysis effective.

7 THREATS TO VALIDITY

A possible threat to internal validity is that VGX may have implementation errors. We mitigate it by inspecting code carefully, doing unit testing when implementing each module, and using a small dataset to test before official experiments. Another threat is that the manual inspection for the success samples may be inaccurate. We mitigate it by writing exploits for some of the samples and confirming results by multiple authors via cross validation.

The main external validity threat is that the dataset we use may not represent real-world vulnerability data distribution. We mitigate it by using manually confirmed real-world dataset like CVE/NVD and removing any dataset overlaps during evaluation.

8 RELATED WORK

Vulnerability dataset curation. SARD [1] and SATE IV [57] are synthetic datasets containing 60K+ vulnerability samples, while BigVul [26] and CVEFixes [15] are real-world datasets containing much less (<10K) samples. FixReverter [68] uses manually derived patterns to inject vulnerabilities to existing code. VulGen [55] injects vulnerabilities by addressing *where* and *how* to inject separately. However, the label precision of these generated datasets is low. In contrast, VGX is the most accurate with >90% precision. Apocalypse [60] uses formal techniques to automatically inject bugs in large software code bases. Fuzzle [37] synthesizes buggy programs by encoding moves in a maze as chains of function calls. In comparison, VGX leverages value-flow-based deep representation learning and human knowledge for vulnerability injection.

Source-code pre-training. Deep learning model pre-training has been widely employed. BERT [24] is a pre-trained model for natural languages. CodeBERT [27], CodeT5 [63], and SPT-Code [51] borrow the idea from BERT to pre-train models for programming languages. In contrast, VGX's pre-training is task-specific, explicitly geared toward our fine-tuning task for vulnerability injection.

Human knowledge integration. Integrating human knowledge into learning-based approaches has been explored. Liu et al. [48] combine GNN with human knowledge to detect smart contract vulnerabilities. ComNet. [32] integrates human knowledge into DL to improve orthogonal receivers. DATGAN [36] integrates DL with expert knowledge to generate tabular data. In comparison, VGX integrates human knowledge into vulnerability injection patterns to boost vulnerability generation.

9 CONCLUSION

We presented VGX, a novel technique for large-scale generation of high-quality vulnerable program samples. VGX generates such samples using vulnerability-introducing code edit patterns. These patterns are initially extracted from real-world vulnerability fixes, augmented by manually defined additional patterns, and diversified through manually derived pattern mutation rules according to human knowledge about real-world vulnerabilities. VGX's design also features a semantics-aware contextualization Transformer to identify right injection contexts, which is customized by value-flow-based position encoding and pre-trained against new objectives to facilitate learning syntactic and contextual structures of code. With this novel design, VGX largely outperforms all of the state-of-the-art peer approaches in terms of the quality of generated samples. We also contribute a large vulnerability dataset resulting from VGX's in-the-wild sample production. We further demonstrated the practical usefulness of this dataset via the substantial improvement it brought to vulnerability detection, localization, and repair, and its ability to help find more real-world vulnerabilities (CVEs).

ACKNOWLEDGMENT

We thank the reviewers for their constructive comments, which helped us improve our paper. This research was supported by the Army Research Office (ARO) through grant W911NF-21-1-0027.

REFERENCES

[1] 2017. SARD: A Software Assurance Reference Dataset. https://samate.nist.gov/SARD/.

[2] 2022. 2022 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html.

[3] 2022. CVE-2017-12991. https://github.com/the-tcpdump-group/tcpdump/commit/50a44b6b8e4f7c127440dbd4239cf571945cc1e7.

[4] 2022. Memory Leak. https://cwe.mitre.org/data/definitions/401.html.

[5] 2022. OpenBSD. https://github.com/bukhalo/openbsd-src/commit/a88c32bfabe8a7fd0b25703230d4adba1d204e0a.

[6] 2022. Race Condition. https://cwe.mitre.org/data/definitions/362.html.

[7] 2022. RawStudio. https://github.com/rawstudio/rawstudio/commit/04cf4f537ffdce5f3e5207bead0ac2d254114cc2.

[8] 2022. Use of Uninitialized Variables. https://cwe.mitre.org/data/definitions/457.html.

[9] 2023. Cybersecurity vulnerability statistics and facts of 2023. https://www.comparitech.com/blog/information-security/cybersecurity-vulnerability-statistics/.

[10] 2023. Data Quality Considerations for Machine Learning Models. https://towardsdatascience.com/data-quality-considerations-for-machine-learning-models-dcbe9cab34cb.

[11] 2023. How Much Data Is Needed For Machine Learning? https://graphite-note.com/how-much-data-is-needed-for-machine-learning.

[12] 2023. The Size and Quality of a Data Set. https://developers.google.com/machine-learning/data-prep/construct/collect/data-size-quality.

[13] Hiralal Agrawal, Richard A DeMillo, R_ Hathaway, William Hsu, Wynne Hsu, Edward W Krauser, Rhonda J Martin, Aditya P Mathur, and Eugene Spafford. 1989. *Design of mutant operators for the C programming language.* Technical Report. Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue.

[14] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.

[15] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*. 30–39.

[16] Yingzhou Bi, Jiangtao Huang, Penghui Liu, and Lianmei Wang. 2023. Benchmarking Software Vulnerability Detection Techniques: A Survey. *arXiv preprint arXiv:2303.16362* (2023).

[17] Piotr Bojanowski, Édouard Grave, Armand Joulin, and Tomáš Mikolov. 2017. Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics* 5 (2017), 135–146.

[18] Haipeng Cai, Yu Nong, Yuzhe Ou, and Feng Chen. 2023. Generating Vulnerable Code via Learning-Based Program Transformations. In *AI Embedded Assurance for Cyber Systems*. Springer, 123–138.

[19] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Transactions on Software Engineering* 48, 09 (2022), 3280–3296.

[20] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2022. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering* 49, 1 (2022), 147–165.

[21] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555* (2014).

[22] Michael L Collard, Michael John Decker, and Jonathan I Maletic. 2013. srcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *2013 IEEE International Conference on Software Maintenance*. 516–519.

[23] Roland Croft, M Ali Babar, and M Mehdi Kholoosi. 2023. Data quality for software vulnerability datasets. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 121–133.

[24] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[25] 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)*.

[26] 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 (MSR)*. 508–512.

[27] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.

[28] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: a transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*. 608–620.

[29] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 935–947.

[30] Xiaoqin Fu and Haipeng Cai. 2019. A dynamic taint analyzer for distributed systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1115–1119.

[31] 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.

[32] Xuanxuan Gao, Shi Jin, Chao-Kai Wen, and Geoffrey Ye Li. 2018. ComNet: Combination of deep learning and expert knowledge in OFDM receivers. *IEEE Communications Letters* 22, 12 (2018), 2627–2630.

[33] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations (ICLR)*.

[34] Jacob A Harer, Onur Ozdemir, Tomo Lazovich, Christopher P Reale, Rebecca L Russell, Louis Y Kim, and Peter Chin. 2018. Learning to repair software vulnerabilities with generative adversarial networks. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 7944–7954.

[35] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. 2022. LineVD: statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*. 596–607.

[36] Gael Lederrey, Tim Hillel, and Michel Bierlaire. 2022. DATGAN: Integrating expert knowledge into deep learning for synthetic tabular data. *arXiv preprint arXiv:2203.03489* (2022).

[37] Haeun Lee, Soomin Kim, and Sang Kil Cha. 2022. Fuzzle: Making a Puzzle for Fuzzers. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[38] Wen Li, Li Li, and Haipeng Cai. 2022. On the vulnerability proneness of multilingual code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 847–859.

[39] Wen Li, Li Li, and Haipeng Cai. 2022. PolyFax: a toolkit for characterizing multi-language software. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE-Demo)*. 1662–1666.

[40] 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)*. 2513–2530.

[41] Wen Li, Jinyang Ruan, Guangbei Yi, Long Cheng, Xiapu Luo, and Haipeng Cai. 2023. PolyFuzz: Holistic Greybox Fuzzing of Multi-Language Systems. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1379–1396.

[42] Wen Li, Haoran Yang, Xiapu Luo, Long Cheng, and Haipeng Cai. 2023. PyRTFuzz: Detecting Bugs in Python Runtimes via Two-Level Collaborative Fuzzing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1645–1659.

[43] 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.

[44] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. 2021. Vuldeelocator: a deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2021), 2821–2837.

[45] 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* 19, 4 (2021), 2244–2258.

[46] 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. In *Network and Distributed System Security (NDSS) Symposium*.

[47] Linus Eriksson. 2022. Tree-Sitter. https://github.com/tree-sitter/tree-sitter.

[48] Zhenguang Liu, Peng Qian, Xiaoyang Wang, Yuan Zhuang, Lin Qiu, and Xun Wang. 2023. Combining Graph Neural Networks With Expert Knowledge for Smart Contract Vulnerability Detection. *IEEE Transactions on Knowledge & Data Engineering* 35, 02 (2023), 1296–1310.

[49] Yisroel Mirsky, George Macon, Michael Brown, Carter Yagemann, Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee. 2023. VulChecker: Graph-based Vulnerability Localization in Source Code. In *USENIX Security Symposium*.

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

[51] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. SPT-code: sequence-to-sequence pre-training for learning source code representations. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. 2006–2018.

[52] 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.

[53] 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.

[54] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2022. Generating realistic vulnerabilities via neural code editing: an empirical study. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1097–1109.

[55] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2023. VulGen: Realistic Vulnerability Generation Via Pattern Mining and Deep Learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2527–2539.

[56] Yu Nong, Rainy Sharma, Abdelwahab Hamou-Lhadj, Xiapu Luo, and Haipeng Cai. 2022. Open science in software engineering: A study on deep learning-based vulnerability detection. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1983–2005.

[57] 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.

[58] Han Peng, Ge Li, Wenhan Wang, Yunfei Zhao, and Zhi Jin. 2021. Integrating tree path in transformer for code representation. *Advances in Neural Information Processing Systems* 34 (2021), 9343–9354.

[59] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

[60] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. 2018. Bug synthesis: Challenging bug-finding tools with deep faults. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 224–234.

[61] Carson D Sestili, William S Snavely, and Nathan M VanHoudnos. 2018. Towards security defect prediction with AI. *arXiv preprint arXiv:1808.09897* (2018).

[62] 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)*. 149–160.

[63] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708.

[64] Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. 2022. VulCNN: An image-inspired scalable vulnerability detection system. In *Proceedings of the 44th International Conference on Software Engineering*. 2365–2376.

[65] Ziyu Yao, Frank F Xu, Pengcheng Yin, Huan Sun, and Graham Neubig. 2021. Learning Structural Edits via Incremental Tree Transformations. In *International Conference on Learning Representations (ICLR)*.

[66] Lechen Yu, Joachim Protze, Oscar Hernandez, and Vivek Sarkar. 2020. A Study of Memory Anomalies in OpenMP Applications. In *International Workshop on OpenMP*. Springer, 328–342.

[67] 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)*. 1099–1103.

[68] Zenong Zhang, Zach Patterson, Michael Hicks, and Shiyi Wei. 2022. FIXREVERTER: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing. In *31st USENIX Security Symposium (USENIX Security 22)*. 3699–3715.

[69] 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)*. 111–120.

[70] Xin Zhou and Rakesh M Verma. 2022. Vulnerability Detection via Multimodal Learning: Datasets and Analysis. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security (AsiaCCS)*. 1225–1227.

[71] 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. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. 10197–10207.

[72] Deqing Zou, Yutao Hu, Wenke Li, Yueming Wu, Haojun Zhao, and Hai Jin. 2022. mVulPreter: A Multi-Granularity Vulnerability Detection System With Interpretations. *IEEE Transactions on Dependable and Secure Computing (TDSC)* 01 (2022), 1–12.