VULGEN: Realistic Vulnerability Generation Via Pattern Mining and Deep Learning

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

2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) | 978-1-6654-5701-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE48619.2023.00211

***Abstract*—Building *new*, powerful data-driven defenses against prevalent software vulnerabilities needs sizable, quality vulnerability datasets, so does large-scale benchmarking of *existing* defense solutions. Automatic data generation would promisingly meet the need, yet there is little work aimed to generate much-needed quality vulnerable samples. Meanwhile, existing similar and adaptable techniques suffer critical limitations for that purpose. In this paper, we present VULGEN, the first injection-based vulnerability-generation technique that is not limited to a particular class of vulnerabilities. VULGEN combines the strengths of deterministic (pattern-based) and probabilistic (deep-learning/DL-based) program transformation approaches while mutually overcoming respective weaknesses. This is achieved through close collaborations between pattern mining/application and DL-based injection localization, which separates the concerns with *how* and *where* to inject. By leveraging large, pretrained programming language modeling and only learning locations, VULGEN mitigates its own needs for quality vulnerability data (for training the localization model). Extensive evaluations show that VULGEN significantly outperforms a state-of-the-art (SOTA) pattern-based peer technique as well as both Transformer- and GNN-based approaches in terms of the percentages of generated samples that are vulnerable and those also exactly matching the ground truth (by 38.0–430.1% and 16.3–158.2%, respectively). The VULGEN-generated samples led to substantial performance improvements for two SOTA DL-based vulnerability detectors (by up to 31.8% higher in F1), close to those brought by the ground-truth real-world samples and much higher than those by the same numbers of existing synthetic samples.**

***Index Terms*—Software vulnerability, data generation, bug injection, pattern mining, deep learning, vulnerability detection**

I. INTRODUCTION

The prevalence of code vulnerabilities are a major cause of security risks with modern software systems [1]. As a response, significant effort has been made in helping secure these systems by detecting [2]–[8] and repairing [9], [10] software vulnerabilities—most of these exemplified works follow data-driven (e.g., machine/deep-learning based) approaches. Indeed, such approaches have been gaining growing momentum in recent years, showing promising performance according to the originally reported experimental results.

Meanwhile, the scarcity of quality vulnerability datatsets has become a critical barrier to further advancing those data-driven defense techniques—without being trained on such datasets, the techniques often fail to perform well in real-world scenarios (e.g., detecting vulnerabilities in large/complex real-world software), just as expected [2], [11]. Not only has it blocked training new, more powerful learning-based approaches, this scarcity is also a main reason behind weak evaluations of existing techniques regardless of their being data-driven or not (e.g., code-analysis-based) [12]–[14]. The urgent need for realistic vulnerability datasets has been put under the spotlight in a recent study [15].

High-quality vulnerability datasets do exist [16]–[19], but they are commonly or even collectively not sizable enough to train powerful data-driven models or serve large-scale benchmarking. On the other hand, existing larger datasets [20], [21] are not well representative of real-world vulnerabilities, while current automated **data-collection** methods suffer from great inaccuracy/noise in the resulting datasets [22]–[24].

To address this data-shortage problem, a few **data-generation** methods have been proposed as well, mainly based on pattern mining/application [25], [26] or machine/deep learning (DL) [15], [27]—the approach in [28] is based on static code analysis and covers one single vulnerability class. Yet these approaches face critical challenges that substantially limit their potentials. In particular, solely pattern mining/application based (i.e., **pattern-based** for brevity) techniques suffer from great ambiguity—the patterns extracted are either too generic or too specific to be applicable, while purely **DL-based** approaches are subject to the data-shortage problem by itself—they need a sizable, quality training dataset to be effective [15] (i.e., the *chicken-egg dilemma*). Like Getafix [25], approaches originally designed for bug [29] or vulnerability repair [10] could be adapted for bug/vulnerability injection as done in [15]. While *conceptually* repair and injection may seem to be dual/reversals to each other, *technically* injection

* Haipeng Cai is the corresponding author.

1558-1225/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE48619.2023.00211

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on May 09,2024 at 03:20:04 UTC from IEEE Xplore. Restrictions apply.

can be harder as it requires more buggy/vulnerable samples which are what we lack (versus repair may just need more normal samples—learning to generate normal code directly without using paired data as in [30], which are richly available). When it comes to vulnerability injection, the problem is even worse given the greater scarcity of vulnerable samples than that of bug datasets.

To tackle these challenges, we propose VulGen, a novel automatic data-generation approach that aims at **realistic injection-based vulnerability generation** via *pattern mining/application collaborating with deep learning in synergy*. With a corpus of existing pairs of vulnerable samples and their fixed versions, VulGen first extracts patterns of vulnerability-introducing code edits which represent **how to inject** vulnerabilities, while training a DL-based model (on the same corpus) to locate **where to inject**. Then, given a normal program, VulGen queries the trained model to identify candidate injection locations where it applies compatible edit patterns to realize vulnerability injection (in the normal program).

The key insight underlying our approach is that, through the *collaboration* of a **deterministic** process (i.e., pattern mining/application) and a **probabilistic** process (i.e., injection localization) and the *separation* of the *how* and *where* parts of injection, VulGen combines the strengths of pattern- and DL-based approaches while mitigating each others' weaknesses. In particular, the localization model guides pattern application to mitigate the ambiguity challenge to pattern-based approaches; meanwhile, the mined patterns inform the localization model to choose the best locations. Also, the chicken-egg dilemma in training the (DL-based) localization model is mitigated by (1) leveraging a large pretrained programming-language model hence reducing general data needs and (2) learning to predict *just the injection locations* without further generating the injected code hence further reducing task-specific data needs (or improving model performance for a given amount of training data since predicting locations alone is intuitively easier than predicting the injected code in addition).

We evaluate VulGen on a real-world vulnerability dataset containing 10,783 pairs of normal functions and respective vulnerable versions for training and testing. VulGen achieved 14.6% **precision** (i.e., percentage of generated samples exactly matching ground truth) and a 69% **success rate** (i.e., percentage of generated samples that are indeed vulnerable). Without an existing peer work targeting vulnerability injection, we adapt a **pattern-based** approach (originally designed for bug repair [25]), a DL/**Transformer-based** text-to-text translation approach (originally for vulnerability repair [10]), and a deep/**GNN-based** neural code editing approach (originally for general code-edits generation [31]) as baselines. VulGen outperforms these potential peer approaches by 16.3–158.2% and 38.0–430.1% in terms of relative precision and success rate improvements, respectively. We also assess the usefulness of the VulGen-generated samples by adding them to the original training sets of two state-of-the-art DL-based vulnerability detectors. The addition boosted their performance significantly (by up to 31.8% greater F1) in both reproduction and replication settings—very close to those by adding the ground-truth vulnerable samples and much higher than adding equal numbers of existing synthetic samples. VulGen is also efficient, generating 900+ vulnerable samples in one hour.

In summary, our paper makes the following contributions:

- To the best of our knowledge, VulGen is the first automatic approach to injection-based *realistic vulnerability generation* without being limited to a particular vulnerability class.
- We show the design of combining *deterministic* and *probabilistic* approaches for program transformation, where *pattern mining/application* and *localization* deal with *how* and *where* to inject respectively while collaborating in synergy.
- We performed extensive experiments that demonstrate significant merits of this VulGen design over both pattern- and learning-based (both sequence and graph modeling) approaches; our results also show substantial improvements the generated samples bring to vulnerability detection.

II. Motivation and Background

In this section, we motivate our vulnerability injection technique using concrete examples and discuss key challenges. Then, we use these examples to illustrate the limitations of existing peer techniques on the challenges. Later, we will use the same examples to illustrate our own approach.

A. Motivation Examples and Challenges

As an example of vulnerable sample generation, let us consider injecting vulnerabilities to existing real-world normal functions. Figure 1 shows three examples of vulnerability injections on normal functions, which delete a buffer size (i.e., `sizeof(d->msg)`) checking for early return, change to use an unsafe memory allocation rather a self-defined, safe one, and remove one of the boundary checking in an `if` condition, respectively. Overall, to inject the vulnerabilities automatically, the technique needs to solve at least two challenges below:

First, the technique needs to know **where** to inject the vulnerabilities. In other code editing tasks, such as bug fixing, the code fragments to be edited have already been provided, either by the testing dataset itself [10] or external static analyzers [25]. However, such location information is not available in the vulnerability generation task, nor is it trivial to obtain. The localization for vulnerability injection needs both *syntactic* (e.g., the code structures) and *semantic* (e.g., information hidden in identifier names and insights into what kinds of statements typically are most prone to vulnerabilities) information in the code [15]. For instance, in the first example in Figure 1, we not only need to locate an `if` statement that contains an early return (syntactic information), but also need to know it is a buffer size checking (semantic information).

Second, the technique needs to know **how** to edit the located code fragments to inject vulnerabilities. For instance, in the second example in Figure 1, the technique needs to change the function call name in the assignment statement from `safe_calloc` to `malloc`. In the third example, the technique needs to remove the OR expression in the `if` condition and replace it with the second expression in the

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on May 09,2024 at 03:20:04 UTC from IEEE Xplore. Restrictions apply.

Delete a buffer size checking

```
static int
cx24116_send_diseqc_msg(/*…*/)
{
  struct cx24116_state *state=fe->demodulator_priv;
  int i, ret;
  if (d->msg_len > sizeof(d->msg))
    return -EINVAL;
  // …
}
```

Use an unsafe memory allocation rather than a self-defined safe one.

```
// …
sz = pdf->xrefs[i].end - ftell(fp);
buf = safe_calloc(sz + 1); => malloc(sz + 1);
SAFE_E(fread(buf, 1, sz, fp), sz,
          "Failed to load /Root.\n");
buf[sz] = '\0';
// …
```

Remove one of the boundary checking in an if condition

```
// …
BDRVVPCState *s = bs->opaque;
uint32_t pagetable_index,pageentry_index;
pagetable_index=offset/s->block_size;
pageentry_index=(offset%s->block_size)/512;
if(pagetable_index >= s->max_table_entries ||
    s->pagetable[pagetable_index]==0xffffffff)
  return -1;
// …
```

Fig. 1. Motivating/illustrating examples on vulnerability injection.

OR expression. These vulnerable functions are not easy to generate, as we need to ensure the syntactic correctness of the code (e.g., any token mistakenly generated in the example would make the code not compilable). Therefore, we need to ensure that the technique edits the code in a correct way.

B. Existing Peer Techniques and Limitations

To motivate our technique, we illustrate two existing peer techniques that can be used for vulnerability injection and discuss their limitations on the two challenges.

1) Getafix: **Getafix** is a technique which automatically fixes bugs using the patterns learned in the existing bug-fixing examples. Specifically, it has three phases to learn the bug-fixing patterns and apply them to fix a new buggy program:

Phase 1: Pattern Mining. Given a set of example fixes where each is a pair of buggy and fixed code, Getafix converts the code into abstract syntax trees (ASTs) which contain a set of nodes indicating the syntactic structure of the code [25]. Then, Getafix employs an AST differencer to get the edits from the buggy code to the fixed code, called concrete edits.

With the concrete edits, Getafix uses *anti-unification* to obtain abstracted edits and then hierarchical clustering to mine the summarized edit patterns for bug fixes. Because of the space limit, we refer readers to the original Getafix paper [25] for further details. After hierarchical clustering, the concrete edit patterns are merged into generalizable edit patterns which can inject vulnerabilities in a variety of normal functions.

Phase 2: Pattern Application. After the hierarchical clustering, there are many edit patterns available, ranging from very generic to very specific [25]. Getafix thus needs to select an appropriate pattern and a candidate pattern-applicable location. It thus grades the pairs of patterns and locations with three scores: (1) the proportion of bugs in the training set that can be fixed by applying the pattern (i.e., *prevalence score*); (2) the proportion of bugs in the training set that can be fixed z lines away from the static analyzer warning location (i.e., *location score*), and (3) the reciprocal of the proportion of subtrees in the given input program that the pattern can match (i.e., *specialization score*) [25]. Then, Getafix ranks the pairs of patterns and locations with the products of the three scores and selects the top-k pairs to apply the patterns to fix bugs.

Phase 3: Validation. In the list of ranked bug fixes, Getafix uses static analyzers to validate whether the bug has been removed. If so, Getafix suggests the bug fixes to developers.

As Getafix uses a deterministic approach to learn bug fixing, the number of training samples can be relatively small compared to the one for DL-based approaches. The hierarchical clustering also allows Getafix to generate human-like (i.e., realistic) bug fixes. Thus, it is a good approach for our vulnerability generation technique to start with. However, it has two major technical limitations for our task:

- Getafix uses static analyzers to locate buggy code to fix, but there is no static analyzer to do so for vulnerability injection.
- The edit patterns mined only match and edit code syntactically without semantic awareness, making the vulnerability injection ambiguous. For example, in the third example of Figure 1, the mined pattern may be like `if(`$h0$`||`$h1$`) => if(`$h1$`)` where $h0$ and $h1$ are place holders that can match any expression. However, such an edit pattern that removes one of the conditions in the `if` statement may not inject a vulnerability if the conditions are not relevant to security. This may make Getafix fail to inject vulnerabilities.

2) Transformer-based Code Edit Model: A number of pretrained transformer models have been built for software engineering (SE) tasks, of which CodeT5 [32] has been shown to be quite promising for semantic-aware code generation [10], [33]. Thus, these models seem to be a good starting point for our task. Yet they also suffer two key challenges:

- Current Transformer-based bug fixing techniques rely on the error messages [33] or the bug location information [10] to be given. Thus, the techniques only need to output the fixed version of the buggy code *fragments* rather than the whole programs/functions. This allows the outputs to be short texts. However, for vulnerability generation, the model has to output the whole programs/functions which are much longer, as we do not have the information that the bug-fixing techniques rely on. Previous work [15] has shown that these models are not good at generating long texts. This greatly limit their ability to directly generate vulnerable code.
- While the CodeT5 model has the capability to understand code semantics, it is not syntax-aware [34]. Since programming languages are highly structured, any erroneous tokens would make the generated code invalid. For example, in the third example of Figure 1, missing or wrongly predicting any token like `->`, `]` or `)` would break the whole program. Thus, every token has to be predicted correctly, but this is difficult for the Transformer model to accomplish *alone*.

III. Overview

Figure 2 gives an overview of our technical design. As shown, VulGen consists of three main technical modules/phases: *pattern mining*, *localization learning*, and *vulnerability injection*, working in two modes. In the mining/learning

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on May 09,2024 at 03:20:04 UTC from IEEE Xplore. Restrictions apply.

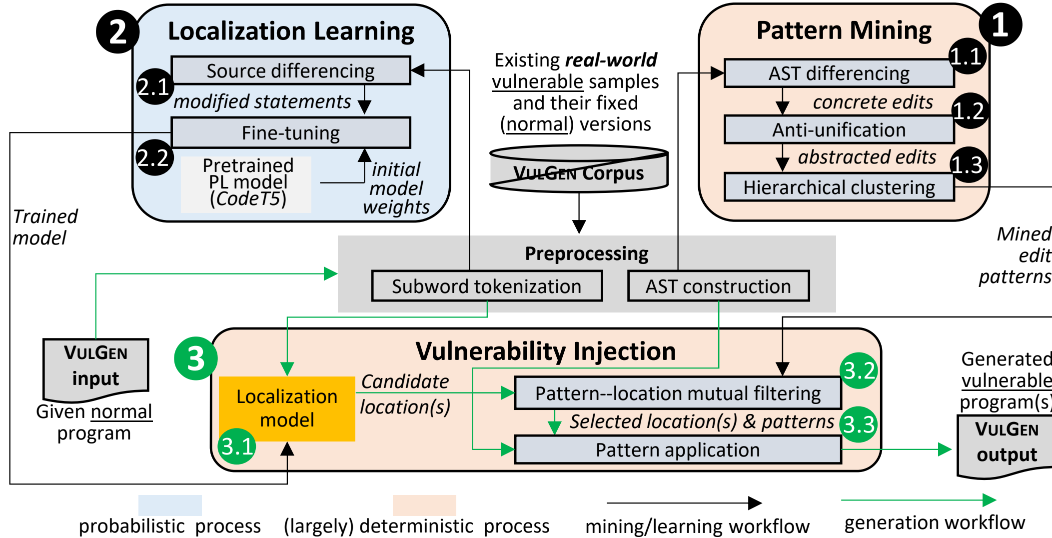

Fig. 2. An overview of VULGEN, including its three key technical modules (phases) and two workflows (modes).

mode, VULGEN mines patterns of real-world vulnerability-inducing code edits (i.e., reversal of respective fixes) from the given corpus of existing vulnerable program samples and their fixed (i.e., normal) versions and then learns to locate where vulnerabilities may be injected. With the resulting edit patterns and the trained localization model, in the subsequent generation mode, VULGEN takes a given normal program as input, queries the model to obtain candidate injection locations and applies compatible patterns, and hence produces vulnerable program(s). Two preprocessing steps, *AST construction* and *subword tokenization*, are shared between both modes.

During the pattern mining phase, VULGEN extracts concrete (AST) edits from the pairs of (normal and respective vulnerable) samples in the given corpus through *AST differencing*, followed by *anti-unification* to obtain abstracted edits and then *hierarchical clustering* to mine the eventual edit patterns—an idea similar to that for bug-fixing pattern extraction in Getafix [25]. Yet the resulting patterns are often either (1) too generic, which are compatible with numerous code locations but not helpful for vulnerability injection since these patterns tend to have many placeholders (holes) that cannot be instantiated, or (2) too specific, which are hardly compatible with any code locations for injection, as illustrated in §II-B1.

Thus, VULGEN comes with localization learning, a dedicated module to disambiguate pattern-based injection. In this phase, VULGEN aims to learn injection localization from real-world developers' historical vulnerability-fixing code-change **locations**. Given the scarcity of such fixes, we leverage CodeT5 [32], a state-of-the-art programming-language *(PL) model that was pretrained* on millions of code samples against relevant objectives. To enable its working for our localization task, VULGEN *fine-tunes* it against those historical locations obtained by *source-level differencing* between the sample pairs in the given corpus. Note that this module learns **only where** to inject (i.e., predicting injection locations), but **not how** (i.e., generating the injected code itself).

Finally, in the vulnerability injection phase, VULGEN feeds the trained *localization model* with the subword-tokenized code of a given input normal program to obtain candidate injection location(s) and selects those compatible with any of the mined edit patterns, followed by injecting vulnerabilities at chose locations via *pattern application*. This results in the vulnerability-injected (i.e., potentially vulnerable) version of the input program. Depending on how many top candidate locations taken from the localization model, VULGEN may produce one or multiple vulnerable programs as its output(s). A key novelty of VULGEN lies in the close collaboration of a **deterministic** process (i.e., pattern mining and application) with a **probabilistic** process (i.e., injection localization), as reflected in the *pattern-location mutual filtering* step: (1) the probabilistic (localization) informs the deterministic (pattern application) about where to apply patterns and which to apply—the resulting locations help filter out incompatible patterns, while (2) the deterministic (pattern mining) helps select the best locations returned by the probabilistic (localization)—the mined patterns help filter out incompatible locations.

The overall vulnerable sample generation (vulnerability injection) process by VULGEN is ***realistic*** for two reasons. First, when an injection is localized, the location prediction is made based on *real-world* vulnerability-introducing code locations. Second, when a vulnerability is injected, the code editing is done by following (the reversal of) how *developers* make vulnerability fixes in diverse *real-world* software projects.

IV. APPROACH

In this section, we describe the detailed design of VULGEN, elaborating on its key technical components/phases shown in Figure 2 and illustrating each using the examples of Figure 1.

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on May 09,2024 at 03:20:04 UTC from IEEE Xplore. Restrictions apply.

A. Pattern Mining

In the pattern mining phase, VulGen follows Getafix to mine the patterns for vulnerability injection. VulGen takes the pairs of normal functions and the respective vulnerable versions as the inputs for pattern mining (as well as for localization learning). Each pair is a vulnerability injection example. To convert the code to ASTs, we use srcML [35], an AST parser for C language, to get the ASTs for each example. We choose srcML because we can convert the parsed AST back to source code easily. Then, VulGen uses GumTree [36] to differentiate the ASTs to get *concrete edits*. Then it follows the original Getafix to perform hierarchical clustering and get the edit patterns for vulnerability injection.

To illustrate, consider the vulnerability-injection case in the second example of Figure 1, VulGen extracts the concrete edit patterns below.

$$safe_calloc(sz+1) => malloc(sz+1) \tag{1}$$

Then, Getafix uses *anti-unification* to merge similar concrete edits into abstracted edits. For example, Pattern 1 and `safe_calloc(bufsize)=>malloc(bufsize)` can be merged into `safe_calloc(`$h0$`)=>malloc(`$h0$`)`, where $h0$ is a placeholder that can match any subtrees. With *anti-unification*, Getafix performs hierarchical clustering to get edit patterns for vulnerability injection.

B. localization Learning

In this phase, VulGen trains a Transformer-based model for locating vulnerability injection spots. We formulate the task as a text-to-text prediction problem: given a normal function, the model generates the statement text where an edit pattern should apply to inject a vulnerability. For instance, in Figure 1, given the whole functions as the input, the model outputs the statements in the red rectangle, without any modification. We use a Transformer-based model only for localization rather than vulnerability injection, because: (1) it is hard for the model to predict whole functions which are long; (2) programming languages are highly structured and thus every token predicted has to be correct, which is hard for the model to achieve.

To build the localization model, VulGen fine-tunes the pretrained Transformer model CodeT5 [32] because CodeT5 is able to understand code semantically. To deal with the out-of-vocabulary (OOV) issue which is widely exists in code relevant tasks [37], we leverage the Byte Pair Encoding (BPE) approach [38] to represent the input and output text. Specifically, BPE splits the original tokens into sequences of characters and merges the frequent symbol pairs into new tokens. Thus, it can split rare tokens into meaningful subwords so that the vocabulary size can be reduced.

Different from traditional Transformer that leverages an absolute positional encoding layer, we use a relative positional encoding layer [39] to captures the relative information between tokens. Specifically, in the relative positional encoding, the self-attention is computed through four matrices: the query matrix Q, the key matrix K, the value matrix V [40], and the matrix P that encodes relative positional information.

$$\text{Attention(Q,K,V,P)} = softmax(\frac{Q(K+P)^T}{\sqrt{d_k}})(V+P) \tag{2}$$

where P is the edge representation for the pairwise inputs. P is supplied as a sub-component of the value matrix [39].

To fine-tune the CodeT5 model for injection localization, we process our examples into a fine-tuning dataset $\mathcal{D} = \{(n_i, s_i)\}_{i=1}^N$, where each sample consists of a normal function n and the respective ground-truth statement s where the vulnerability can be injected, both in text form. Denote the model parameters as θ, the fine-tuning objective is a maximum likelihood estimation which minimizes the following negative log-likelihood loss:

$$L(\mathcal{D};\theta) = \sum_{i=1}^{N} -log\ p(s_i|n_i;\theta) \tag{3}$$

To obtain the ground-truth statement s as our localization target, we apply the `diff` tool to each example and use the modified statement as the ground-truth output. After fine-tuning, the injection localization model is expected to predict a statement to inject a vulnerability in a given normal function.

Given that the injection localization model is a probabilistic model, we leverage beam search to select multiple statements given a normal program. The number of predicted statement candidates replies on a parameter called *beam size* β. Beam search selects the best β statement candidates that have the highest probability. Later we will use different beam sizes to evaluate VulGen and the baseline techniques.

C. Vulnerability Injection

Once we have the edit patterns mined and the localization model trained, VulGen is expected to inject a vulnerability given a new normal function. This input function is first fed into the localization model, and the model locates a statement for vulnerability injection. Since the edit patterns are based on and supposed to be eventually applied back on ASTs, we again use srcML [35] to parse the input normal function into an AST. As the localization model outputs the located statements in source code, we convert all the subtrees in the AST back to source code and compare the source code of each subtree with the localization model output to get the located subtree.

Given the located subtree, we need to select an appropriate pattern to apply. The original Getafix ranks edit patterns based on the three scores described in Section II. Since we cannot use localization score as no static analyzer can be used, we only compute the prevalence score and the specialization score to rank the edit patterns.

Specifically, given an edit pattern E and normal functions n in the training set $\mathcal{D}$, VulGen computes prevalence score for E as follows:

$$s^E_{prevalence} = \frac{|\{n \in \mathcal{D} | E \text{ can inject a vulnerability to } n\}|}{|\mathcal{D}|} \tag{4}$$

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on May 09,2024 at 03:20:04 UTC from IEEE Xplore. Restrictions apply.

This computes the likelyhood that a pattern can inject a vulnerability. Note that in the equation above, we assume that the pattern can find the location to apply perfectly (i.e., once applying E to one of the applicable locations in n can inject a vulnerability, E can inject a vulnerability to n).

Thus, given an edit pattern E and a new normal function n', VULGEN computes the specialization score as:

$$s^{E,n'}_{specialization} = \frac{|\{\text{AST nodes of } n'\}|}{|\{\text{AST subtrees of } n' \text{ that match } E\}|} \quad (5)$$

This avoids the selected patterns to be too general and prioritizes the more specific patterns.

Given an edit pattern E and a new normal function n', VULGEN computes the ranking score $s^{E,n'}_{ranking} = s^{E}_{prevalence} \times s^{E,n'}_{specialization}$ for each pattern and uses the ranking scores to rank the edit patterns. The rationale is to select a pattern that is not only likely to inject vulnerabilities but also specialized enough. Then, VULGEN selects the first pattern in the ranking from higher to lower scores that matches the located statement to inject a vulnerability. However, it is possible that a given function does not involve any security-relevant code hence no vulnerability could be injected. Thus, to reduce false positives, if the located statement cannot match any pattern, VULGEN would not output a function and the input function is identified as "no vulnerability can be injected". Otherwise, the output function is expected to be vulnerable.

To illustrate, consider the third example in Figure 1. Once the localization model locates the `if` statement in the red rectangle, VULGEN ranks the patterns using the ranking score above. Then, the first edit pattern applicable in the ranking like `if(`$h0$`|`$h1$`)`$h2$ `=> if(`$h1$`)`$h2$ will be applied, and the normal function is injected with a vulnerability as shown.

V. EVALUATION

We describe our tool implementation and evaluation dataset, and then seek to answer the following research questions:

- **RQ1**: How effective is VULGEN in vulnerability generation?
- **RQ2**: Hoe does VULGEN compare to a Transformer-based program-transformation approach?
- **RQ3**: How does VULGEN compare to a traditional pattern-based code-generation approach?
- **RQ4**: How does VULGEN compare to a GNN-based code-editing approach?
- **RQ5**: How useful are the VULGEN-generated vulnerabilities for training DL-based vulnerability detectors?
- **RQ6**: How efficient is VULGEN in vulnerability generation?

A. Tool Implementation

As Getafix is not publicly available, we re-implemented its pattern mining and pattern application modules. For localization learning, we use the pre-trained model CodeT5 and the respective APIs provided by HuggingFace [41]. We partly reused the CodeT5 fine-tuning code of VulRepair [10], only for the injection-localization step in VULGEN. As that original code of VulRepair is used for generating vulnerability fixes, we changed/adapted it in order to serve our localization purposes. Our experiments were performed on a server with an AMD Ryzen Threadripper 3970X (3.7GHz) CPU with 32 Cores, an Nvidia GeForce RTX 3090 GPU, and 256GB memory.

TABLE I
EFFECTIVENESS OF VULGEN AND THE BASELINES FOR VULNERABILITY INJECTION. THE NUMBERS IN PARENTHESES INDICATE VULGEN'S RELATIVE IMPROVEMENT COMPARED TO THE BASELINES.

Editor	Precision (Exactly-Matched)	Success Rate
VulGen	14.64%	69%
T5	10.29% (42.27%↑)	17% (305.88%↑)
Getafix	5.67% (158.20%↑)	50% (38.00%↑)
Graph2Edit	12.59% (16.28%↑)	13% (430.77%↑)

B. Dataset

Since the available real-world vulnerability data is rare, we perform a comprehensive literature study for vulnerability analysis datasets and combine them to build a relatively large dataset. As a result, we build our evaluation dataset by including (some of) the vulnerability fixing examples from five reliable human-labeled vulnerability fixing datasets:

1) Devign [2]: 23,355 vulnerability fixing examples in C language from four real-world projects, among which two are publicly available. We include the 7,938 examples from these two available projects to our evaluation dataset.
2) ReVeal [11]: 18,169 human-labeled vulnerable/non-vulnerable samples in C language for vulnerability detection. We select the vulnerable samples that are paired with corresponding fixed versions and finally include 921 vulnerability fixing examples to our evaluation dataset.
3) PatchDB [18]: 12,073 vulnerability fixing examples in C language extracted by nearest link search and then confirmed by humans. We include all of them to our dataset.
4) BigVul [17]: 3,754 vulnerability fixing examples extracted from the CVE/NVD database where 2,185 examples are in C language, we include the 2,185 to our dataset.
5) CVEFixes [19]: 31,092 vulnerability fixing examples from CVE/NVD database where 4,120 are in C language, we include the 4,120 to our dataset.

Thus, we include 27,237 vulnerability fixing examples to our evaluation dataset. All the examples are at function level (i.e., each example is a pair of a vulnerable function and a respective fixed one). We notice that many of the real-world vulnerability fixing edits by the developers not only fix the vulnerabilities themselves, but also modify the functionality or refactor the code, thus there are code changes irrelevant to vulnerability in these examples. To avoid the impact of these irrelevant code changes, we only use samples where the edits are limited to one statement, resulting in 10.783 samples. Since our goal is to inject vulnerabilities, we reverse the fix in each example to get the vulnerability injection examples. The 10,783 vulnerability injection examples are the evaluation dataset we used.

C. RQ1: Effectiveness in Generating Vulnerabilities

We evaluate VULGEN's ability to generate real-world vulnerabilities using the examples in the evaluation dataset. We

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on May 09,2024 at 03:20:04 UTC from IEEE Xplore. Restrictions apply.

TABLE II
DO THE VULGEN-GENERATED VULNERABLE SAMPLES HELP IMPROVE THE DL-BASED VULNERABILITY DETECTORS?

Tool	Setting	Metric	Baseline	Synthetic	Generated	Ground Truth	Wild
Devign	**Reproduction:**	Precision	10.00%	10.69% (6.90%↑)	11.65% (16.50%↑)	10.97% (9.70%↑)	9.74% (-2.60%↑)
	Training: Devign	Recall	26.23%	37.26% (42.05%↑)	52.85% (107.04%↑)	47.91% (101.48%↑)	59.70% (127.60%↑)
	Testing: ReVeal	**F1**	**14.48%**	**16.62% (12.87%↑)**	**19.09% (31.84%↑)**	**17.86% (23.34%↑)**	**16.75% (15.67%↑)**
	Replication:	Precision	9.31%	8.85% (-4.94%↑)	8.59% (-4.18%↑)	8.92% (-4.18%↑)	8.13% (-12.67%↑)
	Training: Devign	Recall	26.56%	24.67% (-7.11%↑)	51.98% (95.71%↑)	54.99% (107.04%↑)	53.48% (101.35%↑)
	Testing: Xen	**F1**	**13.78%**	**13.02% (-5.51%↑)**	**14.75% (7.04%↑)**	**15.36% (11.46%↑)**	**14.12% (2.47%↑)**
ReVeal	**Reproduction:**	Precision	10.63%	12.00% (12.88%↑)	12.00% (12.88%↑)	12.56% (18.16%↑)	11.28% (6.11%↑)
	Training: Devign	Recall	74.90%	49.80% (-33.51%↑)	73.38% (-2.02%↑)	74.52% (-0.51%↑)	79.85% (6.61%↑)
	Testing: ReVeal	**F1**	**18.62%**	**19.35% (3.92%↑)**	**20.63% (10.80%↑)**	**21.50% (15.47%↑)**	**19.77% (6.18%↑)**
	Replication:	Precision	7.68%	6.89% (-10.29%↑)	8.50% (10.68%↑)	8.17% (6.38%↑)	7.91% (2.99%↑)
	Training: Devign	Recall	82.67%	31.64% (-61.72%↑)	60.64% (-26.64%↑)	94.54% (14.36%↑)	95.29% (15.27%↑)
	Testing: Xen	**F1**	**14.05%**	**11.33% (-19.36%↑)**	**14.90% (6.05%↑)**	**15.04% (7.05%↑)**	**14.60% (3.91%↑)**

split the 10,783 examples into 9:1 for training and testing, as prior work did [37]—which also gave us a sizable set (i.e., >1000 samples) for testing. We also checked duplicates between the training and testing sets and removed them to ensure that the two sets have no overlap. As a result, we have 9,704 examples for pattern mining and localization learning. The remaining 1,078 examples are used in the vulnerability injection phase to test the effectiveness of VULGEN. For each testing example, we input the normal function to VULGEN and it outputs a (potential) vulnerable function if it can inject a vulnerability to it. We count the number of output functions that exactly match the ground-truth vulnerable functions, and compute the precision by the proportion of exactly-match functions in the output functions.

With the 1,078 testing examples, VULGEN outputs 963 functions and the remaining 115 are identified as "no vulnerability can be injected" (see Section IV.C). In the 963 functions, 141 of them exactly-match the ground truths. Thus, the precision is 14.64%.

However, it is possible that a vulnerability is injected but the output function does not exactly match the ground truth. This usually happens when the input function has multiple locations to do vulnerability injection. Thus, to further evaluate the effectiveness of VULGEN, we increase the beam size of the localization model to 10. Thus, given a normal function, the model outputs 10 statements for the edit patterns to inject vulnerabilities and a normal function can be used to generate up to 10 functions. With the 1,078 examples, VULGEN generates 9,573 functions. We randomly sample 100 of the generated functions and manually check whether they are vulnerable. Note that 100 is sizable relative to the total number of generated samples as used in our comparison studies. Recent peer work used only <60 samples for similar-purpose manual validation [22]. This is still not ideal, but manually examining a sample is tedious and costly, while general, accurate automated vulnerability detection is unavailable.

The manual checking is done by the first (Rater-1) and second (Rater-2) authors of this paper and a non-author PhD student (Rater-3) who have 2–4 years of experience in software engineering and security, all following the same labeling process. Based on the labels they agreed on, we calculated the inter-rater agreement as 0.7877, 0.7476, and 0.6826 between (Rater-1, Rater-2), (Rater-1, Rater-3), and (Rater-2, Rater-3), respectively, in terms of Cohen's Kappa. These agreements are all substantial, showing reasonable reliability of our manual labeling. Since each generated sample has a corresponding normal sample, they focus on the changed code between the pair and mainly check if the change introduces vulnerabilities (by tracking data/control flow from the changed lines), which also helped mitigate any possible biases during these manual processes. As a result, 69 out of 100 checked functions are vulnerable. Thus, the success rate of vulnerable sample generation estimated by sub-sampling is 69%.

We also assess the generality of VULGEN by examining the vulnerability types of the 100 randomly sampled functions generated and assigning them with CWE vulnerability type IDs [42]. Among them, the success cases of VULGEN covered 18 different CWE IDs, versus those of Getafix, T5, and Graph2Edit only covering 12, 8, and 5 classes, respectively. This indicates that by decoupling *where-to-inject* and *how-to-inject*, VULGEN allows for more flexible/diverse identification of injectable code locations where vulnerabilities can be injected deterministically via pattern matching/application, hence more general vulnerability generation than earlier approaches. Besides, VULGEN learns injection patterns of different vulnerability classes from the training data, making it not limited to generating a particular class of vulnerabilities.

> VULGEN achieves 14.64% exactly-match precision and 69% success rate, suggesting its promising capability for realistic vulnerability generation.

D. RQ2: Comparison with Transformer-based Approach

VULGEN injects vulnerabilities based on the combination of Transformer/CodeT5-based localization and pattern-based code editing. However, as other studies have also shown that the fine-tuned CodeT5 model has the capability to generate the edited code directly [10], [33], people may cast a question: whether we can directly fine-tune the CodeT5 model to generate the vulnerable functions.

To test that, we remove the pattern mining phase and directly fine-tune CodeT5 for vulnerability injection. When fine-tuning CodeT5, we replace the ground-truth outputs, which are

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on May 09,2024 at 03:20:04 UTC from IEEE Xplore. Restrictions apply.

originally the respective statements to inject vulnerabilities, with the respective vulnerable functions. After fine-tuning, the model is expected to output a respective vulnerable function given a normal function. We keep the training examples and testing examples the same as the ones for VULGEN.

Given the 1078 testing examples, the model outputs a new function for each testing example. Thus, the model generates 1078 new functions. However, among the generated 1078 functions, only 111 of them exactly match the ground-truth vulnerable functions, making the precision only 10.29%, where VULGEN outperforms it by 42.27%.

We again increase the beam size to 10 and randomly sample 100 outputs and manually check whether they are indeed vulnerable. Given the 1078 testing examples, the model generate 10780 new functions. In the 100 sampled output functions, only 17 of them are vulnerable and thus the success rate of generating vulnerable functions is only 17%, which is too low to build a high-quality vulnerability dataset.

The failure indicates the limitation of Transformer-based code edit model for vulnerability generation. By manually inspecting the generated samples, we notice that many of the generated functions are not syntactically valid. For example, some generated code ends at the middle of the functions. Some generated code simply repeats the tokens until the maximum limitation of the output tokens. We notice that such failures are more serious when the functions are longer.

> The Transformer-based approach achieves only 10.29% precision and 17% success rate, suggesting its poor capability for vulnerability generation.

E. RQ3: Comparison with Pattern-based Approach (Getafix)

To show the effectiveness of VULGEN's CodeT5-based localization model, we remove the localization model and use the original Getafix approach to inject the vulnerabilities. We directly use the ranked patterns to edit the normal functions to inject vulnerabilities. As the localization score which is based on static analyzer error messages is not available, we cannot rank the statements that the patterns can match. Thus, given a normal function, we first get the top-10 patterns in the ranking and extract the statements that the 10 patterns can match. Then, we randomly select a statement and a pattern that match the statement to inject a vulnerability.

Given the 1078 examples, Getafix generates 1073 new functions. Only 61 of them match their ground-truth vulnerable functions, making the precision 5.67%, where VULGEN outperforms it by 158.20%. To compare the beam size 10 results for VULGEN and the Transformer-based code edit approach, we randomly select 10 pairs of patterns and statements to generate new functions for each given normal sample. If there are less than 10 pairs of patterns and statements, we use all the pairs to generate new functions. Getafix thus generates 8114 new functions. With randomly sampling 100 new functions and manually checking, 50 of them are vulnerable, thus the success rate of generating vulnerable functions is 50%.

The results indicate the limitation of traditional-pattern-based approach for vulnerability injection and show the importance of using semantic-aware model for injection localization. The 5.67% exactly-match precision indicates that, although the mined patterns have the capability to match the statements to inject vulnerabilities syntactically, without the semantic localization, they are difficult to find the correct locations to inject vulnerabilities. However, the 50% success rate for vulnerability injection also indicates the value of traditional pattern-based approach. Compared with Transformer-based code edit approach, it ensures the syntax validity and has the capability to match some special tokens (e.g., memset, free, etc.), although without understanding the context code. This further indicates the necessity to combine semantic-aware approach with traditional-pattern-based approach.

> Getafix achieves 5.67% exactly-match precision, suggesting the necessity of using semantic-aware model for injection localization. However, the 50% success rate also indicates value of pattern-based approach, showing that the combination of pattern-based approach and CodeT5-based localization is promising.

F. RQ4: Comparison to GNN-based Approach (Graph2Edit)

In the study [15], Nong et al. show that the GNN-based code edit approach Graph2Edit [31] achieves the highest effectiveness for vulnerability injection. Graph2Edit takes the AST of a given program as input, converts it into a graph, and use its GNN embedding to predict a sequence of AST edits to generate a new program. The design of the edit operations and the dynamic programming algorithm makes Graph2Edit outperforms other DL-based code editors [15]. Thus, we compare effectiveness of Graph2Edit with VULGEN.

We follow the experiments in [15] to set up Graph2Edit, preprocess our examples into ASTs, use the same examples for VULGEN to train and test the Graph2Edit model. For beam size 1, Graph2Edit generates 1024 new functions and 129 of them exactly match their ground-truth vulnerable functions. Thus, the exactly-match precision is 12.59%, where VULGEN outperforms it by 16.28%. For beam size 10, Graph2Edit generates 10240 functions. Sampling 100 of them and manually checking the 100 functions, only 13 of them are vulnerable, making the success rate only 13%.

The results indicate the advantage of VULGEN compared with the GNN-based approach. We notice that Graph2Edit does not have the capability to understand the code semantic compared with our localization model. One of the main reasons is that our localization model is based on CodeT5 which are trained on millions of code samples. Thus, the fine-tuned model has the capability to deal with more diverse code and is better at understanding the code semantics rather than uses irrelevant code features (e.g., program lengths or single tokens) to edit code. In comparison, Graph2Edit is a randomly initialized model and only the 9704 examples from our dataset are used to train the model. Given the fact that DL models require a large amount data (usually >100,000 samples) to train

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on May 09,2024 at 03:20:04 UTC from IEEE Xplore. Restrictions apply.

the models well, our 9704 examples are too few. Therefore, the Graph2Edit model is overfitted and cannot deal with complex code edit scenarios. Given the fact that there is a lack of vulnerability data to train the DL model, the GNN-based code edit approach may not be suitable for vulnerability injection.

> Graph2Edit achieves 12.59% precision and 13% success rate, indicating its limitation of understanding code semantics compared with VULGEN.

G. RQ5: Usefulness of VULGEN

To evaluate the usefulness of VULGEN, we explore the effectiveness of using the generated functions to improve the DL-based vulnerability detectors. We follow the experiment settings in [15] and perform the evaluation on Devign [2] as well as ReVeal [11] which are the state-of-the-art vulnerability detectors at function level for C language. There might be other even more advanced DL-based detectors (e.g., in the future). Yet our main goal with RQ5 here is to show the improvement our generated samples can bring, instead of the absolute numbers on detection accuracy. Thus, whether the chosen detectors outperform all other options may not be our major concern for this paper. We believe that if our samples can help improve the chosen detectors, they would be expected to help other detectors (e.g., LineVul [37]) as well.

Similar to [15], we apply the *independent testing* that the training samples and testing samples are from different datasets to simulate the real-world vulnerability detection scenario. We use the datasets in [15] as the baseline datasets for training and testing. As our generated functions involve samples from the datasets Devign and ReVeal, we remove the duplicates in the datasets for vulnerability detection. Thus, the numbers of samples in our experiments are different from the ones in [15]. We use Devign dataset (which has 9,744 vulnerable samples and 11,012 non-vulnerable samples) for training and use ReVeal (which has 1,630 vulnerable samples and 16,487 non-vulnerable samples) and Xen (which has 531 vulnerable samples and 7,436 non-vulnerable samples) for testing as the Devign dataset is relatively balanced.

Then, we add the 963 generated functions from VULGEN to the training set of Devign and see whether the new training set improves the performance of the detectors—since we have 963 samples produced by VULGEN, we simply use all of them. To avoid the impact on training brought by any change in the dataset balance (ratio of #vulnerable samples to #non-vulnerable samples), we also add the proportional number of real-world normal samples from [43] to the training set to keep the balance the same as before. Table II column Baseline shows the performance of the two detectors on testing sets ReVeal and Xen using the original Devign training set. The column Generated shows the improvement compared to the baseline using the new training set. We can see that VULGEN's generated functions significantly improve the vulnerability detector performance. For example, in the reproduction setting, our the training set improves Devign's F1 by 31.84%, which generally shows the effectiveness of the generated functions.

To show that the VULGEN's generated functions are better than the synthetic samples, we replace the 963 generated functions with equal number of synthetic vulnerable functions from SARD [20] and re-train the Devign and ReVeal detectors. Table II column Synthetic shows the improvements using the synthetic samples. We notice that the synthetic samples improve the detectors much less than the VULGEN's generated samples do (Reproduction setting for both Devign and ReVeal) or even decrease the performance (Replication setting for Devign and ReVeal). This indicates that the VULGEN's generated functions are more useful than the synthetic ones.

We also compare the VULGEN's generated functions with their respective ground-truth vulnerable functions. Table II column Ground Truth shows the improvements using the ground-truth vulnerable functions of the 963 generated functions. We notice that the improvements are mostly better than the ones of the generated functions (except for Devign's replication setting experiment), which is as expected since our generated functions have 69% success rate and there is noise brought in the rest of generated functions.

To show that VULGEN is not limited to use the normal functions from the examples of vulnerability fixes to generate vulnerable functions, we use the normal functions which are not fixed from vulnerable functions in the BigVul dataset [17] for injection. To support it, and make the experiment comparable, we randomly select the same number (963) of generated functions to improve the training set and column Wild shows the improvements. We notice that although the improvements are less than the ones in the Generated and Ground Truth experiments, they are still better than the ones in the Synthetic experiment, indicating the potential of VULGEN to generate a large amount of useful vulnerable functions.

> VULGEN's generated functions are useful to improve the DL-based vulnerability detectors.

H. RQ6: Efficiency

We measure the efficiency of VULGEN by tracking the time cost of generating the 963 functions. The experiment is performed on the machine we descirbe in Section III.F. We apply 15-process parallel computing for the task. In total, VULGEN takes 55 minutes to generate 963 functions, and thus it generates 17.5 functions per minute in average.

> VULGEN is efficient for vulnerability generation.

VI. DISCUSSION

In this section, we use several case studies to show why VULGEN works better than other learning-based code edit approaches for vulnerability injection.

Pattern-based approach is more effective than Transformer-based model for code editing. Figure 3 shows an example that VULGEN successfully injects a vulnerability but the Transformer-based code edit model does not. The not-commented code is the normal function before edited (lines 1-6 and 16-26). The ground truth of the vulnerability injection

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on May 09,2024 at 03:20:04 UTC from IEEE Xplore. Restrictions apply.

```
1  static int mp_property_video_frame_info(void *ctx,
2          struct m_property *prop, int action, void *arg)
3  {
4      MPContext *mpctx = ctx;
5      struct mp_image *f = mpctx->video_out
6          ? vo_get_current_frame(mpctx->video_out) : NULL;
7  // The code below is added by the translation model
8    /*const char *pict_types[] = {0, "I", "P", "B"};
9      const char *pict_type = f->pict_type >= 1
10         && f->pict_type <= 3
11         ? pict_types[f->pict_type] : NULL;
12     struct m_sub_property props[] = {{"picture-type"/*…*/};
13     MPContext *mpctx = ctx;
14     struct mp_image *f = mpctx->video_out
15         ? vo_get_current_frame(mpctx->video_out) : NULL;*/
16     if (!f)
17         return M_PROPERTY_UNAVAILABLE;
18     const char *pict_types[] = {0, "I", "P", "B"};
19     const char *pict_type = f->pict_type >= 1
20         && f->pict_type <= 3
21         ? pict_types[f->pict_type] : NULL;
22     struct m_sub_property props[] = {{"picture-type"/*…*/};
23 // The statement below is deleted by both VulGen and translation model
24     talloc_free(f);
25     return m_property_read_sub(props, action, arg);
26 }
```

Fig. 3. An example of VULGEN's merits over the Transformer-based approach.

```
1  static inline unsigned int get_rtc_time(struct rtc_time *wtime){
2      struct pdc_tod tod_data;
3      long int days, rem, y;
4      const unsigned short int *ip;
5   // The line below is deleted by VulGen
6      memset(wtime, 0, sizeof(*wtime));
7      if (pdc_tod_read(&tod_data) < 0)
8          return RTC_24H | RTC_BATT_BAD;
9      /*…*/
10     y = 1970;
11  // Graph2Edit modifies the line below is modified into
12  // while (days < 0)
13     while (days < 0 || days >= (__isleap(y) ? 366 : 365))
14     {
15         long int yg = y + days / 365 - (days % 365 < 0);
16         days -= ((yg - y) * 365 + LEAPS_THRU_END_OF(yg - 1)
17                 - LEAPS_THRU_END_OF(y - 1));
18         y = yg;
19     }
20     /*…*/
21     wtime->tm_mday = days + 1;
22     return RTC_24H;
23 }
```

Fig. 4. An example of VULGEN's merits over the GNN-based approach.

is to remove the free statement at line 24 to inject a *memory leak* vulnerability [44]. VULGEN uses its injection localization model to correctly locate that line, and uses the edit pattern to delete the free statement to inject the vulnerability. In comparison, the Transformer-based model also correctly deletes line 24, indicating that it also has the capability to correctly locate the statement to inject a vulnerability. However, it strangely adds the code from line 8 to 15 and completely changes the code functionality. It seems that the translation model wants to delete the statement at lines 16-17 (indeed, deleting the if statement could also inject a *use of null pointer* vulnerability), as the first added 5 lines are the same as the 5 lines after the if statement. Then, it messes up and generates the function body again and deletes the free statement at line 24.

This indicates that the translation model may not be suitable for whole-function code editing. Different from other Transformer-based code edit approaches (e.g., bug repair) that only need to generate a few tokens (e.g., a buggy statement), our vulnerability injection task needs to edit whole functions (of often hundreds of tokens), since we do not have an external static analyzer to extract the statements to inject vulnerabilities and the code contexts are also important. However, the translation model is not good at generating a long sequence of tokens as it needs to generate the tokens one by one iteratively.

Pre-trained CodeT5 model allows for understanding code semantically. Figure 4 shows an example that VULGEN successfully injects a vulnerability but the GNN-based approach Graph2Edit does not. Again, the not-commented code is the normal function before edited. VULGEN successfully generates the ground-truth vulnerable function. It correctly locates the statement at line 6 which initializes a structure pointer and uses its pattern to delete the statement to inject a *use of uninitialized variable* vulnerability, based on the semantic of the statement and the context code. However, Graph2Edit locates to line 13 and uses its sequence of edits to remove the second condition in the while loop. While removing the conditions in while loops may inject vulnerabilities in other cases (e.g., writing bytes one by one to a buffer), this just injects a bug that makes the date on the calendar incorrect.

```
1  static int rt5514_dsp_voice_wake_up_put(struct snd_kcontrol *kcontrol,
2              struct snd_ctl_elem_value *ucontrol)
3  {
4      struct snd_soc_component *component = snd_kcontrol_chip(kcontrol);
5      struct rt5514_priv *rt5514 = snd_soc_component_get_drvdata(component);
6      const struct firmware *fw = NULL;
7      u8 buf[8];
8      if (ucontrol->value.integer.value[0] == rt5514->dsp_enabled)
9          return 0;
10     // The whole if statement below is deleted by Getafix
11     if (snd_soc_component_get_bias_level(component) == SND_SOC_BIAS_OFF)
12     {
13         rt5514->dsp_enabled = ucontrol->value.integer.value[0];
14         if (rt5514->dsp_enabled)
15         {
16             if (rt5514->pdata.dsp_calib_clk_name &&
17                 !IS_ERR(rt5514->dsp_calib_clk))
18             {
19                 /*…*/
20                 // The line below is deleted by VulGen
21                 memset(buf, 0, sizeof(buf));
22                 rt5514->pll3_cal_value = buf[0] | buf[1] << 8
23                                        | buf[2] << 16 | buf[3] << 24;
24                 rt5514_calibration(rt5514, false);
25                 clk_disable_unprepare(rt5514->dsp_calib_clk);
26             }
27             /*…*/
28         }
29         /*…*/
30     }
31     /*…*/
32 }
```

Fig. 5. An example of VULGEN's merits over the pattern-based approach.

The failure of Graph2Edit indicates that it may not have enough capability to understand code semanticsThe main reason of the failure may be the lack of training data (<10K). The DL model of Graph2Edit is not well trained to deal with different code edit scenarios. Thus, fine-tuning a pre-trained code-semantic-aware model is a better way. Previous studies have shown that using pre-trained models like CodeBERT [45] and CodeT5 [32] can improve the DL models for code relevant tasks significantly, although the training datasets may not be very large [10], [37]. VULGEN takes the advantage of the pre-trained model CodeT5 for localization and thus outperforms Graph2Edit for vulnerability injection.

DL-based model allows changing the task for code easily. Figure 5 shows an example that VULGEN correctly locates and deletes the statement at line 21 to inject a vulnerability but Getafix does not. Since we do not have an external static analyzer for localization, once Getafix ranks the top edit patterns to use, it could only randomly select a location to apply the pattern. In Figure 5, Getafix simply matches and deletes an `if` statement where the condition is an equality expression, without other localization information used. Thus, Getafix completely breaks the functionality of the function but

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on May 09,2024 at 03:20:04 UTC from IEEE Xplore. Restrictions apply.

does not inject any vulnerability.

The reason that we cannot use static analyzers for vulnerability injection is that they are mostly rule-based code analyzers. Those rules are defined by human experts based on their experience and knowledge. For example, the static analyzers for bug localization have been studied and developed for many years [46]–[50] and can be directly used for bug localization in the original Getafix. However, the rules are defined by humans and cannot be transferred to our vulnerability injection localization task. In contrast, the DL-based model can be trained for different tasks in the same domain once the model is designed and the training data is ready. The CodeT5 pre-trained model can be fine-tuned for different code relevant tasks such as code summarization, clone detection, and code translation [32]. Thus, we fine-tune it for our vulnerability injection localization task and it makes VULGEN outperforms Getafix for vulnerability injection.

VII. Threats to Validity

Internal validity. As the source code of Getafix [25] is not available, the major threat to internal validity lies in the implementation of the pattern mining and vulnerability injection phases, which might differ from the one in the original Getafix tool. To mitigate this problem, we do unit testing and integration testing on the pattern mining and vulnerability injection phases to ensure they work correctly.

Another threat to validity lies on the hyperparameter setting for training the injection localization model, other baseline models, as well as the DL-based vulnerability detection models. As hyperparamter tuning for these models are expensive, we use the default setting for all the DL models used as they have the best performance in their original evaluation.

External validity. The main threat to the external validity lies in the datasets we used for evaluation. Although the vulnerability fixes in the datasets we include are labeled or confirmed by humans, they cannot be ensured to be precise. Also, many real-world vulnerability fixing examples not only involve the fixing themselves, but also involve other edits not relevant to the vulnerabilities. Thus, the edit patterns we extract and the injection localization training data may have noise. To mitigate this, we only select one-statement-edit vulnerability fixing examples for our evaluation dataset.

VIII. Ethical Implications

As highlighted earlier (§I), we aim to address the data needs for large-scale benchmarking and deep-learning(DL)-based technique development, rather than benefiting/supporting attackers. We expect no ethical concerns because: (1) the generated vulnerable samples are not real-world software and will not be deployed; (2) after being used for training DL-models, these samples will not be disclosed to users—only the trained models are deployed/shared; (3) even if the samples become accessible to attackers who may leverage the vulnerabilities therein against real-world software, such vulnerabilities would be readily detectable by models trained on such samples—and no exploits are provided.

IX. Related Work

Many efforts on building vulnerability datasets exist. SARD [20] and SATE IV [21] are popular datasets which contain over 60,000 vulnerability samples, but the samples are synthetic. BigVul [17] and CVEFixes [19] develop scripts to automatically collect the real-world vulnerability fixing examples based on the reports in the CVE/NVD database [16], but the total numbers of available fixes are still small (e.g, <5,000 fixes for C language). Some other works [22]–[24] develop techniques to automatically detect vulnerability fixes in real-world projects, but the accuracy is low (<60%).

There are also techniques that automatically generate bug/vulnerability data. Zhang et al. [28] develop a framework to automatically generate null-pointer-dereference vulnerability samples, but only one type of vulnerability samples can be generated. FixReverter [26] reverts known bug-fix patterns to inject bugs for benchmarking fuzzers. SemSeed [27] is a technique that seeds realistic bugs semantically using word embedding model, but it can only seed one-line bugs.

Other learning-based techniques that edit code for different purposes exist. Many of them edit code for bugs/vulnerability repair. Getafix [25] is a pattern-based code editor which automatically suggests human-like bug fixes to the developers. VulRepair [10] automatically fixes real-world vulnerabilities using a fine-tuned CodeT5 model [32]. CURE [29] automatically fixes bugs using a code-aware neural machine translation model. However, as vulnerability injection is different from bug/vulnerability repair, these techniques cannot be directly used for vulnerability injection.

In comparison, VULGEN automatically generates vulnerability samples based on the widely available normal samples and the generated samples can be directly used for training vulnerability analysis models without further cleaning.

X. Conclusion

To generate large-scale vulnerability datasets for training or benchmarking vulnerability analysis techniques, we propose VULGEN, the first injection-based vulnerability-generation technique not being limited to one vulnerability class. VULGEN combines the strengths of deterministic (pattern-based) and probabilistic (DL-based) approaches to achieve realistic vulnerability injection. Our evaluation results show that VULGEN significantly outperforms state-of-the-art potential peer techniques for vulnerability injection, and the promising usefulness of the generated samples.

XI. Data Availability

Our source code, datasets, and experimental results are all available in our publicly accessible artifact.

Acknowledgment

We thank the reviewers for their constructive comments which helped us improve our original manuscript. This research was supported by the Army Research Office (ARO, grant number W911NF-21-1-0027), the European Research Council (ERC, grant agreement 851895), and the German Research Foundation within the ConcSys and DeMoCo projects.

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on May 09,2024 at 03:20:04 UTC from IEEE Xplore. Restrictions apply.

References

[1] F. Civaner, "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.

[2] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, 2019.

[3] X. Zhou and R. M. Verma, "Vulnerability detection via multimodal learning: Datasets and analysis," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security (AsiaCCS)*, 2022, pp. 1225–1227.

[4] T. H. M. Le and M. A. Babar, "On the use of fine-grained vulnerable code statements for software vulnerability assessment models," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 621–633.

[5] D. Hin, A. Kan, H. Chen, and M. A. Babar, "LineVD: statement-level vulnerability detection using graph neural networks," in *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 596–607.

[6] X. Fu and H. Cai, "FlowDist:multi-staged refinement-based dynamic information flow analysis for distributed software systems," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2093–2110.

[7] W. Li, J. Ming, X. Luo, and H. Cai, "{PolyCruise}: A {Cross-Language} dynamic information flow analysis," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2513–2530.

[8] W. Li, J. Ruan, G. Yi, L. Cheng, X. Luo, and H. Cai, "PolyFuzz: Holistic greybox fuzzing of multi-language systems," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[9] Z. Chen, S. Kommrusch, and M. Monperrus, "Neural transfer learning for repairing security vulnerabilities in c code," *arXiv preprint arXiv:2104.08308*, 2021.

[10] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "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)*, 2022, pp. 935–947.

[11] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering (TSE)*, 2021.

[12] Y. Nong, H. Cai, P. Ye, L. Li, and F. Chen, "Evaluating and comparing memory error vulnerability detectors," *Information and Software Technology*, vol. 137, p. 106614, 2021.

[13] Y. Nong and H. Cai, "A preliminary study on open-source memory vulnerability detectors," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 557–561.

[14] Y. Nong, R. Sharma, A. Hamou-Lhadj, X. Luo, and H. Cai, "Open science in software engineering: A study on deep learning-based vulnerability detection," *IEEE Transactions on Software Engineering (TSE)*, 2022.

[15] Y. Nong, Y. Ou, M. Pradel, F. Chen, and H. Cai, "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*, 2022, pp. 1097–1109.

[16] H. Booth, D. Rike, G. A. Witte *et al.*, "The national vulnerability database (NVD): Overview," 2013.

[17] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A c/c++ code vulnerability dataset with code changes and cve summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*, 2020, pp. 508–512.

[18] X. Wang, S. Wang, P. Feng, K. Sun, and S. Jajodia, "Patchdb: A large-scale security patch dataset," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021, pp. 149–160.

[19] G. Bhandari, A. Naseer, and L. Moonen, "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)*, 2021, pp. 30–39.

[20] P. E. Black *et al.*, "SARD: A software assurance reference dataset," in *Anonymous Cybersecurity Innovation Forum.()*, 2017.

[21] V. Okun, A. Delaitre, P. E. Black *et al.*, "Report on the static analysis tool exposition (sate) iv," *NIST Special Publication*, vol. 500, p. 297, 2013.

[22] Y. Zheng, S. Pujar, B. Lewis, L. Buratti, E. Epstein, B. Yang, J. Laredo, A. Morari, and Z. Su, "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)*, 2021, pp. 111–120.

[23] W. Li, L. Li, and H. Cai, "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)*, 2022, pp. 1662–1666.

[24] T. Fehrer, R. C. Lozoya, A. Sabetta, D. Di Nucci, and D. A. Tamburri, "Detecting security fixes in open-source repositories using static code analyzers," *arXiv preprint arXiv:2105.03346*, 2021.

[25] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–27, 2019.

[26] Z. Zhang, Z. Patterson, M. Hicks, and S. Wei, "FIXREVERTER: A realistic bug injection methodology for benchmarking fuzz testing," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3699–3715.

[27] J. Patra and M. Pradel, "Semantic bug seeding: a learning-based approach for creating realistic bugs," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021, pp. 906–918.

[28] S. Zhang, "A framework of vulnerable code dataset generation by open-source injection," in *2021 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*, 2021, pp. 1099–1103.

[29] N. Jiang, T. Lutellier, and L. Tan, "CURE: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1161–1173.

[30] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022, pp. 959–971.

[31] Z. Yao, F. F. Xu, P. Yin, H. Sun, and G. Neubig, "Learning structural edits via incremental tree transformations," *arXiv preprint arXiv:2101.12087*, 2021.

[32] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[33] B. Berabi, J. He, V. Raychev, and M. Vechev, "Tfix: Learning to fix coding errors with a text-to-text transformer," in *International Conference on Machine Learning*. PMLR, 2021, pp. 780–791.

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on May 09,2024 at 03:20:04 UTC from IEEE Xplore. Restrictions apply.

[34] C. Niu, C. Li, V. Ng, J. Ge, L. Huang, and B. Luo, "SPT-code: sequence-to-sequence pre-training for learning source code representations," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 2006–2018.

[35] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration," in *2013 IEEE International Conference on Software Maintenance*, 2013, pp. 516–519.

[36] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (ASE)*, 2014, pp. 313–324.

[37] M. Fu and C. Tantithamthavorn, "LineVul: a transformer-based line-level vulnerability prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 608–620.

[38] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," *arXiv preprint arXiv:1508.07909*, 2015.

[39] P. Shaw, J. Uszkoreit, and A. Vaswani, "Self-attention with relative position representations," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT), Volume 2 (Short Papers)*, 2018, pp. 464–468.

[40] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017.

[41] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, "Huggingface's transformers: State-of-the-art natural language processing," *arXiv preprint arXiv:1910.03771*, 2019.

[42] S. Christey, J. Kenderdine, J. Mazella, and B. Miles, "Common weakness enumeration," *Mitre Corporation*, 2013.

[43] G. Lin, W. Xiao, J. Zhang, and Y. Xiang, "Deep learning-based vulnerable function detection: A benchmark," in *International Conference on Information and Communications Security*. Springer, 2019, pp. 219–232.

[44] W. Li, H. Cai, Y. Sui, and D. Manz, "PCA: memory leak detection using partial call-path analysis," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE-Demo)*, 2020, pp. 1621–1625.

[45] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[46] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan, "Building useful program analysis tools using an extensible Java compiler," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2012, pp. 14–23.

[47] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, "Moving fast with software verification," in *NASA Formal Methods Symposium*. Springer, 2015, pp. 3–11.

[48] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA)*, 2004, pp. 132–136.

[49] D. Kroening and M. Tautschnig, "CBMC–c bounded model checker," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2014, pp. 389–391.

[50] D. Marjamäki, "Cppcheck: a tool for static c/c++ code analysis," https://cppcheck.sourceforge.io/, 2013.

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on May 09,2024 at 03:20:04 UTC from IEEE Xplore. Restrictions apply.