2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-6654-3786-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/SANER53432.2022.00114

VELVET: a noVel Ensemble Learning approach to automatically locate VulnErable sTatements

Yangruibo Ding*, Sahil Suneja†, Yunhui Zheng†, Jim Laredo†, Alessandro Morari†, Gail Kaiser*, Baishakhi Ray*
*Columbia University, †IBM Research

***Abstract*—Automatically locating vulnerable statements in source code is crucial to assure software security and alleviate developers' debugging efforts. This becomes even more important in today's software ecosystem, where vulnerable code can flow easily and unwittingly within and across software repositories like GitHub. Across such millions of lines of code, traditional static and dynamic approaches struggle to scale. Although existing machine-learning-based approaches look promising in such a setting, most work detects vulnerable code at a higher granularity – at the method or file level. Thus, developers still need to inspect a significant amount of code to locate the vulnerable statement(s) that need to be fixed.**

This paper presents VELVET, a novel *ensemble learning* approach to locate vulnerable statements. Our model combines graph-based and sequence-based neural networks to successfully capture the local and global context of a program graph and effectively understand code semantics and vulnerable patterns. To study VELVET's effectiveness, we use an off-the-shelf synthetic dataset and a recently published real-world dataset. In the static analysis setting, where vulnerable functions are not detected in advance, VELVET achieves 4.5× better performance than the baseline static analyzers on the real-world data. For the isolated vulnerability localization task, where we assume the vulnerability of a function is known while the specific vulnerable statement is unknown, we compare VELVET with several neural networks that also attend to local and global context of code. VELVET achieves 99.6% and 43.6% top-1 accuracy over synthetic data and real-world data, respectively, outperforming the baseline deep learning models by 5.3-29.0%.

***Index Terms*—Security Bugs, Vulnerability Localization, Ensemble Learning, Transformer Model, Graph Neural Network**

I. INTRODUCTION

Rapid detection and elimination of vulnerabilities is crucial to protect production software from malicious attacks. Unfortunately, the shortcomings of traditional program analysis and software testing techniques become apparent at the scale of the software nowadays [1]–[3]. For example, dynamic analysis tools are known to suffer from high false negatives, as they cannot reach many code regions, particularly given the huge size of modern applications and infrastructure. Static analysis tools scale better but require configuration with known vulnerability patterns (i.e., rules), typically running behind the attackers, and tend to report high false positives.

Recent progress in AI techniques, combined with the availability of large volumes of source code, presents an opportunity for security analysts to apply data-driven approaches that augment traditional program analysis. Researchers have explored applying deep-learning techniques to identify security vulnerabilities [4]–[14]. These works typically learn vulnerability patterns from large amounts of vulnerable/non-vulnerable examples without active manual effort. However, previous approaches are mostly limited to predicting vulnerable methods or files, without *locating* the statement that really triggers the vulnerability. Such coarse-grained vulnerability detection slows down developers seeking to locate and fix a vulnerability, since they still need to spend significant debugging effort to inspect hundreds or even thousands of lines of source code manually.

However, it is challenging to locate vulnerabilities at the finer granularity of identifying vulnerable statements. First, existing vulnerability detection tools [4], [6], [12] classify the function as a whole, and a recent research study [13] revealed that these tools learn high-level vulnerable features and cannot highlight the individual vulnerable statements. In contrast, localization requires the model to learn more concrete statement-level vulnerable features; the model needs to pay attention not only to the individual statements but also to the control flows and data dependencies among them. Second, manually annotating vulnerable statements requires significant effort, so collecting a large volume of reliable training data containing vulnerable location information is expensive. We address these challenges by (i) developing a novel *ensemble learning* approach, VELVET, that learns to capture code semantics at statement granularity from both local and global context. (ii) pre-training on large amounts of synthetic data to learn artificial vulnerability patterns, and then fine-tuning on a smaller real-world dataset, which enables the model to understand more complex patterns even though large real-world annotated datasets are not available.

Modeling Vulnerability Localization. We propose VELVET to locate vulnerable statements. Our design stems from two insights: (i) the model needs to capture the semantics of the vulnerable statements, and (ii) the semantics often depend on both local and global context. To this end, VELVET consists of two main steps:

(i) Learning Node Semantics. For locating a vulnerable statement, it is important to understand the statement semantics (*e.g.,* control and data dependency, context, *etc.*). In a static analysis setting, such semantics can be captured well with a code graph, where each graph node represents code elements and edges represent the dependencies between the nodes. Representing these dependencies via a graph has proven effective to understand the code syntax and semantics by many previous studies [4], [12], [13], [15]–[21]. In this work, we use a Code Property Graph (CPG) [22] to represent the code. We then

978-1-6654-3786-8/22/$31.00 ©2022 IEEE
DOI 10.1109/SANER53432.2022.00114

use a Deep Learning (DL) model to learn node semantics from the CPG. The DL model essentially learns a node-level embedding that captures the node content along with contextual dependencies. We then feed the embedded node representation to a classifier to classify the node as vulnerable or not. We further map the node back to the corresponding statement as the final prediction.

(ii) Incorporating Global and Local Context with Ensemble Learning. To locate the vulnerable statement, developers spend significant amounts of time debugging the program, trying to both understand the functionality at a high level and, also, estimate the behaviors from local context of suspicious code blocks. Listing 1 shows a confirmed CVE from our real-world dataset. To identify the out-of-array access at line 146, developers need to know the latest update of `block_ptr`, which is in the surrounding context (line 138), together with the faraway declarations of variable `row_ptr` and `pixel_ptr`. Thus, to locate the vulnerable line, developers need to reason about both *local* and *global* context. To mimic real-world debugging practice, we propose a novel ensemble approach to learn both contexts for vulnerability localization.

Using our VELVET framework, we capture node semantics with a transformer-based model and a GGNN model. We use a linear layer and a softmax to assign vulnerability probabilities to each embedded node. The node with the highest vulnerability probability will be marked as vulnerable, and will be mapped back to the source code statement. Our ablation study shows that *transformer is better able to capture long-range dependencies, i.e., global context, while GGNN captures short-range local dependencies better*. Thus, we combine these two models as an ensemble for final localization.

Pre-training & Fine-tuning. Another challenge for DL-based vulnerability localization tools is the scarcity of high-quality, real-world vulnerability datasets with well-annotated statement-level information, *i.e.,* which specific line or lines are the triggers of the security issue (*e.g.,*, line 146 in Listing 1). Fortunately, a recently launched real-world vulnerability dataset, D2A [23], contains precise vulnerability location information. We apply D2A as the main resource on which to build the prototype for our data-driven technique. To imitate the practical scenario at best, we sort the functions in D2A based on their commit dates. We train the model on the *past* commits and evaluate the performance on the *latest* ones.

Because of the time-consuming data collection process and the expensive manual validation, D2A has a relatively small size and is not enough to sufficiently train a generalized model. One possible alternative is to train a localization model on synthetic vulnerabilities: for example, NIST's Juliet Test Suite [24] was artificially produced to imitate CWE [25] vulnerability patterns, and it indicates the location information for each sample. However, models trained on synthetic data do not usually perform very well in real scenarios [13]. In this work, we propose a practical mitigation to address these concerns: we pre-train VELVET with large-scale synthetic data, forcing the model to first learn simple, artificial patterns, and then fine-tune with D2A examples to capture the complexity of real-world patterns. We show that the pre-training and fine-tuning workflow mitigates the dataset scarcity problem.

Results. In the static analysis setting, where vulnerable functions are not detected in advance, VELVET achieves 2.7× and 4.5× better performance than baseline static analyzers on the synthetic and real-world dataset separately, and significantly reduces the false positives and false negatives. For the vulnerability localization task, where we assume the vulnerable function has already been perfectly detected, VELVET achieves 99.6% accuracy on the NIST Juliet Test Suite [24]. With further fine-tuning on the D2A dataset, VELVET achieves 43.6/63.9% localization accuracies for top-1/3 predictions over the much more complex real-world data. Our ablation study further provides evidence that Transformer is better at capturing distant dependencies while GGNN focuses better on local context of statements, and our VELVET's ensemble approach is the most effective way of combining global and local information for vulnerability localization – outperforming the baseline deep-learning models by 5.3%-29.0%.

Listing 1: Out-of-array accesses vulnerability: CVE-2013-7009

```
// project: ffmpeg (commit sha: 920046a)
// file: libavcodec/rpza.c
1  static void rpza_decode_stream(...)
2  {
14 unsigned short *pixels = ...;
15 int row_ptr = 0;
16 int pixel_ptr = 0; // Fix: int pixel_ptr = -4;
   ...
135  case 0x00:
136    if (s->size - stream_ptr < 16)
137      return;
138    block_ptr = row_ptr + pixel_ptr;
   ...
146    pixels[block_ptr] = colorA;
147    block_ptr ++;
148    }
149    block_ptr += row_inc;
162}
```

This paper makes the following contributions:

- We propose an ensemble neural architecture, VELVET, to locate vulnerabilities at statement-level by successfully capturing local and global program dependencies [26].
- We design the model learning process as pre-training on the synthetic data, JULIET, and fine-tuning on real-world data, D2A, following a practical workflow that alleviates data inadequacy concerns regarding real-world vulnerabilities.
- We evaluate VELVET on both JULIET and D2A. Our ablation studies show our ensemble approach significantly reduces false positives and false negatives and performs 4.5× better than the baseline static analyzers on real-world data. We also show VELVET's ensemble approach is the most effective way to capture vulnerable contexts among baseline deep-learning models.

II. BACKGROUND

Graph Neural Network (GNN). A graph is a common way to represent source code, where each code element is modeled as a graph node, and relations between the code elements are captured by the edges. Graph Neural Network (GNN) is a deep-learning model that learns directly from

graph structure. GNN has been applied to bug detection [12], [13], [27] and fixing [17], [18], [20], [28]. GNN learns the node representations by aggregating the information through the graph structure where nodes can only communicate via neighboring edges. Thus, after sufficient training, each node gains knowledge about its *local* neighborhood. In this way, GNN leans more fine-granular information about semantic (*e.g.,*, data-dependencies) and syntactic (*e.g.,*, tree structure) properties of code than mere token-based representations [15]. In this work, we implement GGNN [29] as representative of graph-based models.

Transformer Model. Transformer model [30] is the state-of-art model for sequence learning and has proven to be effective for source code modeling tasks [17], [31]–[35]. Transformer leverages the power of attention mechanisms (*i.e.,* self-attention and encoder-decoder attention) and builds the architecture entirely on that basis. Multi-head self-attention enables the model to attend to any set of code elements across arbitrarily long distances so that each element will maintain a *global* [17] view over the entire sequence. Thus, compared to GNN, Transformer can aggregate information from two faraway nodes more efficiently. Nevertheless, the Transformer model usually treats code as a sequence of tokens (keywords, variable, *etc.*) and ignores the underlying graph structure of programs. Theoretically, the multi-head attention is able to capture the relations between code tokens during training, but in practice, learning "edges", *de novo*, is more challenging than explicitly defining them with graph structure in advance.

Ensemble Learning. Various neural networks (*e.g.,* GNN and Transformer) are designed out of divergent instincts and thus have divergent architectures. With the random initialization of model weights, different neural architectures can learn quite distinct aspects of the same dataset, even for the same task. This variance leaves challenges for a single model to capture the complexity of the data distributions. Ensemble learning provides a practical solution to minimize individual architectures' noisy bias and improve overall performance: it aggregates multiple models' decisions by either asking models to vote or (weighted) averaging each model's predictions. Such a combination can build a more stable neural network with a comprehensive understanding of the whole data. The ensemble method has shown its effectiveness in code modeling tasks, *e.g.,* Lutellier *et al.* [33] use ensemble learning to repair bugs and report better results than a single-model counterpart.

Due to the diversity of vulnerability patterns, one single model will struggle to generalize when locating vulnerabilities. Instead, combining the knowledge learned by multiple neural architectures can be more effective in identifying diverse patterns. To this end, we propose an ensemble approach that integrates the individual advances of GNN and Transformer to predict the vulnerable statements.

III. Approach

This section presents our VELVET framework, which aims to predict the vulnerable statements in source code. VELVET is built on the premise that similar vulnerabilities have occurred in the past [36], [37] and can be learned from the previous experience. In particular, VELVET analyzes the code as graphs and tries to identify vulnerable graph nodes.

To this end, we train VELVET on vulnerable/non-vulnerable samples in a supervised learning setting, where graph nodes are annotated with a vulnerability probability—true vulnerable nodes are annotated with 1 and non-vulnerable nodes are marked with 0 probability. We then design the vulnerability localization as a classification task where each node is assigned a vulnerability probability. The node with the maximum vulnerability probability will be predicted as the vulnerable location. With sufficient training, we expect the model to learn the patterns of where vulnerabilities are likely to occur, transplant this knowledge to unseen samples with similar characteristics, and make predictions accordingly. Limited by the availability of annotated datasets, we currently only consider functions with a single vulnerable statement/node. However, it should be straightforward to extend VELVET to locate multiple statements, which we plan as future work once we get such datasets.

A. VELVET's Localization Framework

Locating the vulnerable node in a graph can be regarded as classifying graph nodes as vulnerable/non-vulnerable. Thus, we define the vulnerability localization problem with respect to a code graph, where we assume a program can be directly transformed to a graph structure (Section III-B). We consider a code graph as a set of nodes and edges $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ indicates the set of nodes of the graph and $\mathcal{E}$ represents the list of edges connecting the nodes. Given a graph $\mathcal{G}$, the goal for the vulnerability localization task is to locate the vulnerable node $v \in \mathcal{V}$ by predicting its index, $y \in |\mathcal{V}|$, where $|\mathcal{V}|$ represents the number of nodes in the graph. Therefore, we build the samples in our dataset as $\mathcal{D} = \{\mathcal{G}_i, y_i\}^m$, where m is the size of our dataset.

Given a graph $\mathcal{G}$, this involves three main steps:

(i) Learn the semantic representation of each node of $\mathcal{G}$ in an embedded space;
(ii) Use the node-level semantic embedding to assign a vulnerability probability to each node; and
(iii) Select the node with highest vulnerable probability as the vulnerable location of $\mathcal{G}$.

To identify non-vulnerable graphs, we inject a dummy node to every graph. This dummy node acts as a general representation for the entire graph and indicates whether this graph contains vulnerabilities or not. If a sample is non-vulnerable, we mark the dummy node as the ground-truth location.

We design a node embedding method ϕ (Section III-B) to vectorize the nodes v_i as $h_i \in \mathbb{R}^d$, where d is the embedding dimension. The set of h_i that contains all the node embeddings of a graph is defined as $\mathcal{H}$. Given a sample $\{\mathcal{G}, y\}$, we then feed the vectorized graph into a neural-network model to learn the semantic node representation $\mathcal{H}' = F_{model}(\mathcal{H}, \mathcal{E})$.

A well-trained model is expected to encode sufficient semantic knowledge of a node, together with its context, so we feed the transformed semantic representation, h' of each node

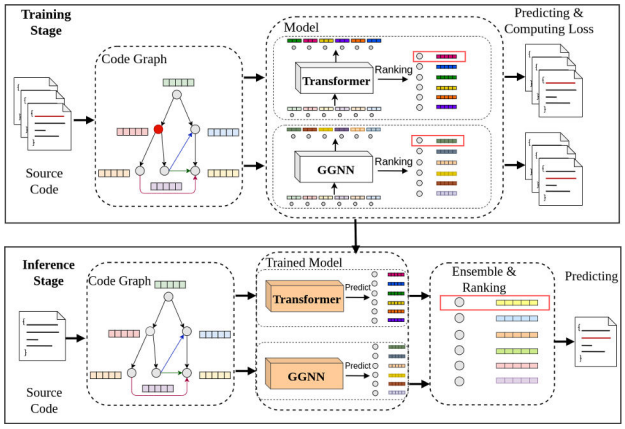

Fig. 1: **VELVET Overview**

to a classifier that tries to compute the probability of each node being vulnerable. More formally, the learned $\mathcal{H}' \in \mathbb{R}^{d_h \times |\mathcal{V}|}$ (d_h is the hidden dimension of the model) is fed into a linear layer with trainable weights $\mathbf{W} \in \mathbb{R}^{d_h \times 1}$ and bias $\mathbf{b} \in \mathbb{R}^{1 \times |\mathcal{V}|}$ to get a score for each node and then a softmax layer for probability $\mathbf{P} = softmax(\mathbf{W}^\top \mathcal{H}' + \mathbf{b})$. Given the vulnerable probability of each node, we take the index of the most probable node as the prediction $\hat{y} = argmax_{i \in |\mathcal{V}|}(p_i)$, and compute cross-entropy loss comparing with the ground-truth.

Figure 1 illustrates the end-to-end workflow of the proposed approach. Similar to many NN-based tools, our technique contains two stages: Training and Inference. The following sections explain the details of each module of the framework.

B. Building Vectorized Code Graphs

Graph Vectorization. In our work, we use Code Property Graph (CPG) [22] to represent the graph semantics of programs (code graph module in Fig. 1). CPG is a code representation designed specifically for vulnerability detection. Besides AST, CFG, DFG edges, it also includes several other types of edges that encode detailed information regarding program dependencies. Specifically, we use *Joern* [38] to generate CPG. To vectorize the node information, we train a *word2vec* model over all possible code tokens in our dataset, and for each node, we concatenate the token embeddings corresponding to that node, as the initial node representation.

Vulnerable Node Annotation. In the graph representation, we annotate the AST node that corresponds to the vulnerable statement as the ground-truth. As shown in Figure 1, the red line in the training sample represents the vulnerable statement, and we annotate the corresponding (red) node in the graph as the ground truth.

C. Locating Vulnerable Nodes with Ensemble Model

As shown in the example of Listing 1, to identify the integer overflow statement, a localization tool should maintain a global view to understand the initialized value of `pixel_ptr` from its declaration, together with the local context about the latest updates of the variable `block_ptr`'s value. Capturing such a combination of long-range and short-range dependencies is necessary for the localizer to successfully identify the vulnerable statement. To this end, we propose our ensemble approach built on two main components to capture these two distinct dependencies: GGNN and Transformer.

Graph-based neural networks are effective at understanding the semantic order of programs, since they directly learn control flows and data dependencies with the pre-defined edges. However, training involves a message passing algorithm where nodes only communicate with their neighbors. The ability to learn long-range dependencies is limited by the number of message passing iterations, which are typically set to a small number (*e.g.,* less than eight) due to computational cost [4], [12], [13], [15], [17], [39]. Such a limitation will result in an inherently *local* [17] model. In contrast, Transformer allows *global*, program-wise information aggregation [17], and without pre-defined edges, the self-attention mechanism of Transformer is expected to encode considerable code semantics – which can be complementary to those defined explicitly by the code graph. Therefore, to learn the diversity of vulnerable patterns, we *separately* train these two distinct models and use their predictions in an *ensemble learning* setting at inference time (Training Stage in Fig. 1).

Local GGNN Model. As the input, the initial vectorized node representation, $\mathcal{H}$, is passed to the GGNN model, along with the list of edges $\mathcal{E}$. Following the design of Li *et al.* [29], the GGNN model aggregates information from node-neighbors using message passing over different edge types with distinct weights. This results in an aggregation vector, representing the information from the neighborhood, for each node at time-step $t-1$. Each node is then, at time-step t, updated by aggregating its own representation of $t-1$ and its neighbors' information using GRU units. After K time steps, the output of GGNN model will be the transformed node representations, encoded with *local* context for each node $H^g = [h_1^g, h_2^g, ..., h_{|\mathcal{V}|}^g]$

Global Transformer Model. In order to learn the implicit code dependencies complementary to GGNN, we only provide Transformer with AST nodes of the graph as a sequence. Removing edges can also enforce the model to learn the long-range dependencies in a more efficient way, since it does not have to reach faraway node step-by-step. We align the AST nodes in the order that reserves the original source code's sequential logic. For example, if the source code has a order of statements as: {`stmt1 -> stmt2`}, then when building the input sequence, all the nodes belonging to `stmt1` will be placed at the left side of all those belonging to `stmt2`. With the power of multi-head self-attention [30], the final node representations transformed by the model are encoded with a *global* view over the whole function [17] $H^{tr} = [h_1^{tr}, h_2^{tr}, ..., h_{|\mathcal{V}|}^{tr}]$

The final node representations from the model (either GGNN or Transformer) are then fed into a multilayer perceptron layer to get a single score for each node: $S^g = [s_1^g, s_2^g, ..., s_{|\mathcal{V}|}^g] \in \mathbb{R}^{|\mathcal{V}|}$ for GGNN, $S^{tr} = [s_1^{tr}, s_2^{tr}, ..., s_{|\mathcal{V}|}^{tr}] \in \mathbb{R}^{|\mathcal{V}|}$ for Transformer. During training, all vulnerability scores are passed into a softmax layer to get the per-node vulner-

ability probability, and calculate the cross-entropy loss. We note that GGNN and Transformer are trained independently, and each model does its back-propagation and updates weights without interaction.

D. Ensemble Learning

As shown in Figure 1, in the Training Stagr we independently train two distinct neural architectures, GGNN and Transformer, to force them to learn the diverse aspects of vulnerable patterns. In the Inference Stage, we aim at combining the knowledge of both models to make comprehensive predictions on previously unseen data. To this end, we propose an ensemble approach to aggregate the predictions from both models. Concretely, as shown in Figure 1, we input the vectorized code graph into both well-trained models, and they will output the transformed node representations H^g, H^{tr} (Section III-C) and then the vulnerability scores, S^g and S^{tr} are computed. To aggregate the predictions, we calculate ensemble vulnerability scores, S^{en}, for all nodes by averaging: $S^{en} = 0.5 * S^g + 0.5 * S^{tr}$, and then the node with the highest ensemble score will be the predicted vulnerable node. We note that the averaging for ensemble scores can be weighted, but further heuristics need to be introduced to decide the weights for different models.

The ensemble approach is technically straight-forward, but works quite well in practice. In Section V, we show the effectiveness of VELVET's ensemble and empirically demonstrate that GGNN and Transformer models are indeed learning diverse aspects of vulnerable patterns, as expected intuitively.

IV. STUDY DESIGN

In this section, we present our datasets and how we train and test the models with them using distinct evaluation metrics.

A. Datasets

We conducted experiments with the two datasets, JULIET (a synthetic dataset) and D2A (a real-world dataset), which include statement-level vulnerability annotations. Concretely, each sample is a function, and the vulnerable location is the statement that directly exposes the vulnerability (*e.g.,* Line 146 in Listing 1). Every sample in the dataset has one single vulnerable statement and is written in the C language. Note that, there are some other existing C-language vulnerability datasets [6], [12], [13], [40] that annotate vulnerabilities only at the function level. Since our goal is to localize vulnerabilities at the statement level, we cannot use them.

Synthetic Dataset (JULIET): The JULIET Test Suite [24] is a synthetic dataset containing intentional flaws to test static analysis tools. The test cases in the dataset have vulnerabilities covering 118 different Common Weakness Enumeration (CWE) classes [25] and well-annotated location information. We extracted around 24,000 vulnerable functions with corresponding faulty locations, and also 26,000 non-vulnerable functions without any vulnerable statements.

Although synthetic datasets provide large amounts of data to train large models [6]–[8], [41], they fail to capture the complexity of real-world data and tend to restrain the model to simple vulnerable patterns [13]. To overcome this issue, we also apply the recently published D2A dataset.

Real-world Dataset (D2A): It is challenging and expensive to collect real-world vulnerabilities with well-annotated statement-level information, so such datasets are rare. Zheng *et al.* recently published the D2A dataset [23] which they collected from multiple C/C++ projects such as `OpenSSL` and `LibTIFF`, which are core components of cloud services. These projects have also been studied by many existing security-related works [21], [42], but without providing the annotations we need. Zheng *et al.* collected vulnerabilities using static program analysis and differential analysis. Compared to existing datasets, D2A preserves more details such as function-call traces and locations that trigger the vulnerabilities. The authors also spent much manual effort to ensure the accuracy of D2A's labels and location information.

In this work, we derive our real-world dataset from D2A, and we end up with approximately 2,500 unique functions with annotated vulnerable locations and 2,500 non-vulnerable counterparts in total. Our dataset contains 9 main vulnerability types covering 18 CWE types. Table I shows the details. The longest function spans 1269 lines of source code, where it would be very challenging for developers to locate the vulnerable statement even if they know the function is problematic.

Even though both datasets have roughly the same number of vulnerable and non-vulnerable functions, our work focuses on statement-level localization, and at the statement granularity both datasets are actually super **imbalanced**, aligning well with the relative rareness of vulnerabilities [13], [43].

TABLE I: Distribution of vulnerability types in the datasets. Column 1&2 show how the vulnerability types in D2A map to the corresponding CWEs in JULIET. Column 3&4 show the respective proportions of the type occurrences. Noted that JULIET covers more vulnerability types than D2A.

D2A Type	JULIET Type	JULIET (%)	D2A (%)
Integer Overflow	CWE190-191, 194-197	21.9%	56.3%
Buffer Overrun	CWE121-122, 124-127	25.0%	30.8%
NullPtr Dereference	CWE476	0.7%	4.9%
Memory Leak	CWE401	2.0%	0.9%
Dead Store	CWE563	1.0%	1.3%
Divide by Zero	CWE369	1.6%	0.3%
Null Dereference	CWE690	1.9%	3.0%
Uninitialized Value	CWE457	1.6%	1.9%
Use After Free	CWE416	0.4%	0.6%

B. Model Pre-Training & Fine-Tuning

a) Migration From the Synthetic to the Real-world. As mentioned in Section IV-A, our real-world dataset, D2A, is much smaller than JULIET due to the high cost of collecting and annotating the locations of real-world vulnerable statements. Given D2A's small size, the usual same-dataset train-test scheme would likely cause the models to overfit.

In this work, we adopt a workflow designed to alleviate this concern of data inadequacy by *pre-training* VELVET on a large amount of synthetic data and *fine-tune* it on the

limited real-world data. We first sufficiently train models on synthetic data with a relatively large learning rate and then fine-tune the pre-trained models on real-world data with a smaller learning rate. The intuition behind this design is: the ample synthetic data has already exposed a portion of the practical vulnerability distribution, so we ask models to first understand this part with adequate samples, and then keep learning other (more complex) parts when fed with what would otherwise be an inadequate set of real samples. In Section V-D, we experimentally determine that models successfully leverage this "pre-training + fine-tuning" setting to migrate the knowledge learned from synthetic data to the real-world scenario.

b) Training, Validation, and Testing Split. In the synthetic dataset, JULIET, we split the whole dataset into three parts, TRAIN/VALID/TEST, with a ratio of 90%:5%:5%. We train VELVET on the TRAIN split from scratch for 10 epochs and evaluate the performance on TEST split.

For D2A, to imitate the practical scenario at best where we learn from previous vulnerabilities, we sort the functions based on their commit dates. We then *train the model on the past commits and evaluate the performance on the latest ones*. Again, we split the sorted samples into TRAIN/VALID/TEST with the ratio of 90%:5%:5% (TEST split is made up of the latest 5%). We fine-tune the trained models from the synthetic dataset on TRAIN split for 50 epochs and evaluate on TEST split. We note that the numbers of epochs for training are carefully selected: we ensure that, in such settings, the validation loss on VALID split of all the models will no longer decrease for 3 consecutive epochs.

c) Experimental Configuration Models are configured with the input hidden dimension of 256, a dropout rate of 0.1. The learning rate for pre-training is 10^{-4}, and for fine-tuning is 10^{-5}. We use Gated Graph Neural Network (GGNN) model [29] with 8 time steps, similarly to several code modeling works [4], [12], [13], [15], [17], [18]. For the Transformer, we use 6 encoder layers, 8 attention heads, while the attention dimension is 512 and feedforward layers of dimension 2048. All models are implemented using Tensorflow 2.2.0, CUDA 10.1, and trained using 8GB Nvidia GeForce RTX 2080 SUPER GPU. For data processing, our machine takes 26.8 seconds to generate every 1000 CPGs, and takes 60.9 seconds to vectorize them. It takes 32 minutes to train word2vec embeddings. The model training on JULIET takes 14 hours, and 24 hours on D2A. These numbers are at par with existing deep-learning-based vulnerability analysis [12], [13], [17].

C. Evaluation of Statement-level Localization

As we introduced in Section III-A, the model prediction will be the index of a graph node. To evaluate the localization performance in a practical scenario, we maintain a mapping between a node and its corresponding line number in the source code. We map both the ground-truth node and the predicted node to their source line number and check whether they match. We evaluate statement-level localization performance using the following metrics:

Top-k Accuracy. We use top-k accuracy as one metric to evaluate the localization quality. We map the top-k predicted nodes with the highest scores to their source line numbers, and if at least one of the predictions matches the ground-truth, we define the prediction as correct.

Prediction Distance. While accuracy is an important metric, it fails to fully capture realistic aspects of the vulnerability localization problem. For instance, when a wrongly predicted location is not too far from the true target, the developer can still find the vulnerability by glancing at the surrounding code. In order to evaluate this aspect, we measure how far the prediction is from the true location. We define the distance between these two as *Prediction Distance* and we calculate it as $Distance = |l_{pred} - l_{true}|$, where l_{pred} is the line number of the model prediction, and l_{true} is the ground-truth.

D. Baselines

Static Analyzers One major motivation of deep-learning-based vulnerability detection tools is to overcome the drawbacks of rule-based static analyzers. We use them as baselines to show the improvement brought by VELVET. We pick three open-source static analyzers, Infer [44], FlawFinder [45] and RATS [46]. These are popular and frequently used by developers and researchers. For example, Infer is widely used by large enterprises [23], [47] and FlawFinder is integrated into many open-source code scanners such as GitHub Code Scanner [48] and Codacy Security Scan [49]. These three static analyzers are also used as baselines by work on deep-learning-based vulnerability detectors [6], [7], [14], [41].

Comparing Architectures under VELVET's framework To figure out the best neural architecture that can be fit into our framework for vulnerability localization, we compare different models by replacing the architecture in the model module in Figure 1. For example, we compare the ensemble model with GGNN-only and Transformer-only models in Section V-A and V-B, resp. Note that the transformer here is a bit different than the commonly used transformer of CodeBert [35], Roberta [50] *etc.*, where the inputs are token sequences. In our case, the input consists of pre-processed graph nodes, as we use node embedding instead of token embedding.

Comparing Aggregation Strategies under VELVET's framework VELVET aggregates the information learnt from *global* and *local* contexts as an ensemble. Other researchers have tried different aggregation strategies: Hellendoorn *et al.* [17] propose Graph-Sandwich models, which stack sequence-based model and graph-based model to capture the *global* and *local* semantics for detecting and fixing variable-misuse bugs. They also propose GREAT, which integrates edge information of code graphs into the Transformer model. Dinella *et al.* [18] propose a graph-based model to learn bug-fixing edits, where they incorporate a *global* pointer mechanism to locate the buggy node. These previous works share similar insights of aggregating information from diverse contexts, although for different tasks. We adapted these methods in our VELVET's framework to compare different ways to aggregate *global* and *local* information for vulnerability

localization. We implement Transformer-sandwich, GREAT, and the localization part of Hoppity by reusing their open-source packages [17], [18]. To compare under identical settings, we replace the model part in Figure 1 with the different architectures and keep all the other parts as-is.

V. Results

We evaluate VELVET with the following research questions:

- **RQ1: Evaluating Classification & Localization.** Can VELVET locate vulnerabilities if they are not detected in advance?
- **RQ2: Evaluating Localization.** How does VELVET perform on fine-grained vulnerability localization?
- **RQ3: Evaluating Ensemble Strategy.** Is the ensemble model an effective way to combine global and local context, compared with existing methods?
- **RQ4: Evaluating Fine-tuning Design.** What are the contributions of the VELVET's pre-training & fine-tuning design?

A. RQ1: Evaluating Classification & Localization

Motivation. First, we check how well VELVET can locate vulnerable statements in a most realistic setting, where we do not know in advance whether the functions are vulnerable. This mimics a typical static analysis setting, where the static analyzer aims to find vulnerable lines. This experiment can be thought of as Classification & Localization, where VELVET will classify a function as vulnerable/non-vulnerable and then locate the statement within the vulnerable function.

In this first RQ, we focus on the comparison between VELVET and the rule-based static analyzer baselines. We prioritize this discussion, since we aim at leveraging data-driven approaches to improve the quality of static analyzers in general, to further alleviate developers' debugging efforts. To comprehensively evaluate the effectiveness of our approach, we compare three variants of VELVET with static analyzers: VELVET-ENSEMBLE, VELVET-GGNN, VELVET-TRANSFORMER.

Methodology. As mentioned in Section III-A, to fit non-vulnerable functions into our framework, we add a dummy node with index of 0 to every sample regardless of its vulnerability. If the sample is non-vulnerable, then the ground-truth will be 0, the index of the dummy node; otherwise the ground-truth will still be the index of the vulnerable node. Thus we successfully fit the vulnerability localization and the non-vulnerable function identification tasks into the same multi-class classification framework. We evaluate this task in:

1) *Multi-class prediction*: The most intuitive metric is class-prediction accuracy in the multi-class classification setting, and we refer to it as *Prediction Accuracy* for further discussion. This setting is specific to our node-based classification and not applicable to static analyzers.
2) *Vulnerability Classification*: To evaluate VELVET's ability to detect vulnerabilities at function level, we define the predictions that point to the dummy node as non-vulnerable and otherwise as vulnerable. Note this setting is coarse-grained—if a vulnerable node has ground-truth $x(x > 0)$, but the model predicts non-dummy node $y(y > 0)$, where $x \neq y$, we still regard the prediction as correct in this setting. Thus, this setting basically evaluates whether there is a vulnerable node anywhere in the function body.
3) *Vulnerability Localization*: This is evaluated by top-1 accuracy, as we discussed in Section IV-C, where we evaluate VELVET's ability to correctly predict vulnerable statements. We will not discuss this accuracy for RATS, since it tends to identify the root-cause location of a vulnerability (*e.g.,* line 16 of Listing 1), yet our datasets annotate the location where the vulnerability occurs (*e.g.,* line 146 of Listing 1). Thus, RATS has very bad accuracy (nearly 0%) on our datasets. On the contrary, we confirmed that Infer and FlawFinder share our approach to location annotation.

Result: VELVET significantly outperforms static analyzers, reducing false positives and false negatives. Table II shows the results when training the model on the combination of vulnerable and non-vulnerable functions. All variants under our VELVET framework outperform static analyzers by a large margin, in three settings and on both datasets. The best-performing VELVET-ENSEMBLE achieves 99.6% localization accuracy on JULIET and 30.1% on D2A, while the rule-based static analyzers, at best, only achieve 36.9% and 6.7%, respectively. Our data-driven approach also reported many fewer false positives (FP) and false negatives(FN) compared with the static analyzers. In the function-level vulnerability detection setting, the precision/recall/f1 of the VELVET variations are significantly higher than all baseline static analyzers. The results empirically validate that VELVET can help alleviate developers' concerns regarding the FP/FN issues of rule-based static analyzers.

We further analyze VELVET's neural architecture module to figure out the best model for static analysis. Compared with the single GGNN and Transformer, the ensemble approach wins for *Prediction Accuracy*, F1 and localization accuracy by a clear margin. The results, in general, reveal the effectiveness of the ensemble approach for combining two distinct architectures. Interestingly, we notice that Transformer beats the ensemble model by less than 1% for classification accuracy. This is likely because the global view of the Transformer model enables it to have better performance on the more general vulnerability detection task, but this advantage is diluted by the local GGNN model when combining the two models. This leaves us an interesting question about how to improve the ensemble methodology when two models make contradictory predictions. We discuss a solution in Section V-C.

B. RQ2: Evaluating Localization

Motivation. After realizing the advantages of VELVET, we are curious about the best-suited neural architecture for our primary goal, vulnerability localization. Thus, we further isolate this task by checking the efficiency of VELVET variants only on the vulnerable samples. In other words, we check how efficiently VELVET can identify a vulnerable statement given the function containing the statement is known to have a vulnerability. This mimics the scenario that by using some other off-the-shelf tools (*e.g.,* Devign [12]), we already know

TABLE II: Results when jointly training models on vulnerable and non-vulnerable functions. As discussed in Section V-A, Pred Acc. is the *Prediction Accuracy*, Vul-CLS indicates the *Vulnerability Classification* setting, and Vul-LOC Acc. is the top-1 localization accuracy.

Approach	JULIET						D2A					
	Pred	Vul-CLS				Vul-LOC	Pred	Vul-CLS				Vul-LOC
	Acc.	Acc.	Precision	Recall	F1	Acc.	Acc.	Acc.	Precision	Recall	F1	Acc.
VELVET-ENSEMBLE	**99.5%**	**99.6%**	**99.9%**	**99.3%**	**99.6%**	**99.6%**	**51.1%**	58.9%	**76.2%**	48.1%	**59.0%**	**30.1%**
VELVET-GGNN	93.6%	94.2%	**99.9%**	89.0%	94.2%	98.5%	45.5%	56.7%	73.2%	39.1%	51.0%	19.6%
VELVET-TRANSFORMER	99.3%	**99.6%**	**99.9%**	**99.3%**	**99.6%**	99.1%	47.2%	**59.3%**	70.5%	**50.4%**	58.8%	29.3%
Infer*	N/A	69.4%	41.5%	54.4%	47.1%	36.9%	N/A	N/A	N/A	N/A	N/A	N/A
FlawFinder	N/A	58.3%	36.6%	51.0%	42.6%	8.6%	N/A	48.7%	53.3%	6.0%	10.7%	6.7%
RATS	N/A	59.3%	36.5%	46.4%	40.9%	N/A	N/A	45.8%	47.9%	16.4%	24.2%	N/A

*We do not compare with Infer on D2A, since Zheng *et al.* [23] used Infer during data collection. Thus, it is unfair to compare with Infer on D2A.

which functions are vulnerable. However, a C/C++ function may contain hundreds of lines of code, so it requires significant human effort to pinpoint the vulnerable location before attempting to fix it. To this end, for this RQ we only train the model on the vulnerable functions of our datasets.

Result-A: Ensemble model shows the best performance on localization. As shown in Table III, the ensemble approach wins against the single models, GGNN and Transformer, on both datasets for different metrics. For the JULIET dataset, the benefits are not as pronounced since learning vulnerable patterns in this synthetic dataset is a relatively easy task for all models. For D2A, the difference is more noticeable: by incorporating both global and local context learned by the two different architectures, VELVET-ENSEMBLE improves the top-1 accuracy of single VELVET-GGNN by around 29.0% and VELVET-TRANSFORMER by 9.5%. We also check the vulnerability types and find VELVET is effective in locating integer overflow, buffer overrun, and null-pointer deferences, which aligns with the dominant types in the training data (Table I). We also notice that the localization accuracy increases compared with those in Table II when isolating the localization task. The reason is, such an isolated training setting enables models to focus on learning the vulnerable triggering locations and thus decrease the overall difficulty of learning to predict the vulnerable nodes. This implies that if a fairly good function vulnerability detector is available, training a localizer with only vulnerable samples will be a better choice.

TABLE III: Test performance of vulnerability localization under VELVET framework. Top-k represents the localization accuracy for top-k predictions. Distance represents the average *Prediction Distance* between the prediction and the ground-truth.

VELVET	Juliet		D2A		
	Top-1	Distance	Top-1	Top-3	Distance
Ensemble	**99.6%**	**0.04**	**43.6%**	**63.9%**	**7.0**
GGNN	98.1%	0.10	33.8%	54.9%	9.7
Transformer	99.4%	0.06	39.8%	62.4%	8.0

Further, VELVET gives us a decent *Prediction Distance*, indicating that the vulnerabilities can be located within a 7-line window around the ground-truth vulnerable statements. The ensemble setting not only beats the baselines but, overall, it is a respectable scope-reducer for debugging as the average length of a D2A function is 80.3 lines and a significant fraction of D2A functions have hundreds of source lines.

Result-B: Transformer learns the global context, while GGNN captures the local context. We have empirically demonstrated that global and local context together can improve vulnerability localization. We further investigate the performance of VELVET-GGNN and VELVET-TRANSFORMER on the 133 (most recent) vulnerabilities in D2A-TEST — *GGNN and Transformer are manifesting complementary capacities to localize the vulnerability*. Due to the repetition and simplicity of certain vulnerable patterns, two models locate 37 vulnerabilities in common; besides these, GGNN can further locate 8 individual vulnerabilities and Transformer alone can locate 16.

We inspect the concrete samples that one model correctly predicts but the other fails. We compute the average function length of the model-specific correct predictions. As expected, Transformer's correct predictions have a larger function length than for GGNN: GGNN's correct predictions include functions with, on average, 27.75 lines and 138.0 graph nodes; for Transformer's correct predictions, however, the functions include 48.38 lines and 288.6 graph nodes, on average, which are 74.3% longer in lines and 109.1% larger in nodes than GGNN. We further conduct statistical test to make sure that the longer average value is not caused by extremely large outliers. This result supports our intuition that Transformer has a better *global* view and is more effective in processing larger code graphs, whereas GGNN focuses more on the *local* contexts of code snippets. We also show two concrete vulnerabilities in D2A-TEST: Listing 2&3. Both can be located by the ensemble approach, but the single model fails on one of them. Listing 2 has a larger size and the global contexts of `out`, `mtmp` and `outlen` are necessary to identify buffer overflow. In contrast, Listing 3 is much smaller and the variable dependencies are mostly in the surrounding lines.

Listing 2: Buffer Overrun vulnerability from D2A. VELVET-ENSEMBLE and VELVET-TRANSFORMER can localize this vulnerability at line 42 while VELVET-GGNN fails.

```
// project: openssl (commit sha: 0211740)
// file: crypto/ec/ecdh_kdf.c
1  int ecdh_KDF_X9_63(...)
2  {...
18 for (i = 1;; i++) {
19   unsigned char mtmp[EVP_MAX_MD_SIZE];
     ...
32   if (outlen >= mdlen) {
33   if (!EVP_DigestFinal(mctx, out, NULL))
34     goto err;
35   outlen -= mdlen;
36   if (outlen == 0)
37     break
```

```
    ...
42   memcpy(out, mtmp, outlen); ...
69 }
```

Listing 3: Integer Overflow vulnerability from D2A. VELVET-ENSEMBLE and VELVET-GGNN can localize this vulnerability at line 19 while VELVET-TRANSFORMER fails.

```
// project: ffmpeg
// sha: 4bd869e
// libavcodec/aacdec.c
1  static int read_audio_mux_element(...)
2  {
3    int err;
4    uint8_t use_same_mux = get_bits(gb, 1);
     ...
14   if (latmctx->audio_mux_version_A == 0) {
15     int mux_slot_length_bytes =
     read_payload_length_info(latmctx, gb);
     ...
19     else if (mux_slot_length_bytes * 8 + 256 <
     get_bits_left(gb)) { ...
25   return 0;
26 }
```

C. RQ3: Evaluating Ensemble Strategy

Motivation. We have shown the effectiveness of the ensemble model for comprehensively understanding the *global* and *local* contexts, compared with the single model. However, the ensemble approach is not the only way to achieve this aggregation for code modeling tasks, as discussed in Section IV-D. We incorporate the alternate aggregation strategies to better evaluate the ensemble approach. We train the models in two settings: first, we only include the vulnerable samples to reveal the localization performance (Vul-only), and then we expand the data to contain both vulnerable and non-vulnerable functions (Hybrid). To evaluate performance, we use the *Prediction Accuracy* defined in Section V-A.

TABLE IV: Comparison of *Prediction Accuracy* between our ensemble method and the existing models.

Model	JULIET		D2A	
	Vul-only	Hybrid	Vul-only	Hybrid
VELVET-ENSEMBLE	**99.6%**	**99.5%**	**43.6%**	**51.1%**
VELVET-GGNN	98.1%	93.6%	33.8%	45.5%
VELVET-TRANSFORMER	99.4%	99.3%	39.8%	47.2%
VELVET-TRANSSANDWICH	99.2%	99.0%	41.4%	49.8%
VELVET-GREAT	99.1%	98.8%	36.8%	41.6%
VELVET-HOPPITY	98.8%	98.1%	35.3%	32.5%

Result-A: VELVET's ensemble strategy outperforms existing models that aggregate global and local contexts. As shown in Table IV, the ensemble approach manifests a more powerful learning capacity compared with the existing models we studied. VELVET-ENSEMBLE beats all other competitors for both vulnerability-only and hybrid settings on both datasets. Specifically, on the real-world D2A, VELVET-ENSEMBLE illustrates the generalization of the ensemble approach to understand the complicated real-world vulnerable patterns, compared with the three re-implemented existing works. This result shows empirically that the ensemble model is a more direct and effective way to combine *global* and *local* knowledge than stacking distinct models, since the stack of models will still share learned insights during training, which may prevent them from learning more diversity. However, compared with the single GGNN and Transformer, the Transformer-sandwich model still reports overall better results with a clear margin on D2A. This also suggests the significance of incorporating distinct model designs to capture varied aspects of code.

Result-B: Ensemble of more models alleviates the disagreement among models and further improves performance. Our main goal is to showcase the advantages of ensemble learning with local and global information, so we initially use one global model and one local model to conceptually illustrate the effectiveness. However, as mentioned in Section V-A, Transformer and GGNN sometimes make contradictory predictions and the dominant model is not always correct. Consequently, such disagreements harm the ensemble's performance. We provide a solution to alleviate this concern: adding more models of the same architecture to the ensemble. As a proof-of-concept, we enlarge VELVET-ENSEMBLE to contain *two* Transformer and *two* GGNN models by varying the initialization seeds, and we compare the 4-model variant of VELVET with the 2-model one in both Vul-only and Hybrid settings on D2A. Table V shows 4-model VELVET is a clear winner and it can also beat all baselines in previous RQs. The results further show the effectiveness of the ensemble approach and indicate VELVET's potential practical deployment.

TABLE V: Performance of adding one more GGNN and one more Transformer model to the VELVET-ENSEMBLE. The newly added models are randomly initialized and have exactly the same configuration with the original ones.

VELVET	D2A Vul-Only			D2A Hybrid		
	Vul-LOC Acc.			Pred	Vul-CLS	Vul-LOC
	Top-1	Top-3	Distance	Acc.	Acc.	Acc.
2-model	43.6%	63.9%	7.0	51.1%	58.9%	30.1%
4-model	**45.9%**	**68.4%**	**6.5**	**52.4%**	**60.6%**	**31.6%**

D. RQ4: Evaluating Fine-Tuning Design

Motivation. In Section IV, we introduced the setting of applying fine-tuning based on the pre-trained JULIET models to real-world data, to try to address the data inadequacy problem. In this RQ, we evaluate the rationality and advantages of this design. Theoretically, we expect the pre-trained JULIET models to already understand a portion of the real-world vulnerable patterns, since the synthetic data is designed to imitate the practical scenario, even as it struggles with the complexity of the real-world distribution. To understand whether this is the case, we evaluate the well-trained JULIET models of VELVET-ENSEMBLE directly on D2A to see how much knowledge has been directly transferred from the synthetic data. Again, we use the two same settings, Vul-only and Hybrid, as in Section V-C for this evaluation.

Result: Pre-training on synthetic data and fine-tuning on real-world data effectively mitigates the real-world vulnerability samples inadequacy problem. As shown in the "Before" row of Table VI, the pre-trained JULIET model is already able to correctly localize 12% of D2A test samples, with an average prediction distance of 14.8 in the Vul-only setting, and has 16.5% *Prediction Accuracy* for the Hybrid

TABLE VI: Results of VELVET-ENSEMBLE before and after the fine-tuning on two datasets, with two settings. The "Before" row means we directly evaluate the well-trained JULIET model on D2A. The "After" row means the model's performance after the sufficient fine-tuning on real-world data.

Fine-Tuning	Vul-only		Hybrid		
	Vul-LOC ACC	Distance	Pred ACC	Vul-CLS ACC	Vul-LOC ACC
Before	12.0%	14.8	16.5%	56.3%	3.8%
After	43.6%	7.0	51.1%	58.9%	30.1%

setting. This result shows that a significant portion of real-world vulnerable patterns are already well-understood by the pre-trained JULIET models, providing a great start for fine-tuning. Also, with its knowledge of tens of thousands of synthetic samples, the model is less likely to be overfitted to the relatively small D2A dataset. As a comparison, we also show the performance after the fine-tuning (*i.e.,* "After" row in Table VI). We can see that the fine-tuning significantly improves the performance on the real-world dataset, and even with just 2.5k vulnerable samples and 2.5k non-vulnerable samples, the model can show a generalized result in both Vul-only and Hybrid settings. The results also provide solid evidence to the rationality of our mitigation of the data inadequacy, which hopefully can help researchers move forward without being too worried about small real-world dataset sizes for data-driven approaches.

VI. RELATED WORK

Fault Localization. Locating buggy statements with the available test cases has been well-studied for decades [51]–[60]. Spectrum-based (SBFL) and mutation-based (MBFL) fault localization are two well-known approaches. SBFL takes each statement's coverage information and its test case results as input, calculates a suspiciousness score, and ranks the statements' scores to indicate the most buggy one(s). MBFL mutates the code by pre-defined rules to evaluate each statement's actual effects on the pass/fail outcomes of test cases. VELVET is not directly comparable, since we focus on source code without test cases and coverage reports.

DL-based Vulnerability Detection. To automatically learn vulnerable features and patterns directly from source code, recent work [4], [6]–[8], [12]–[14], [27] applies a wide variety of deep-learning models. However, most of this work predicts function-level vulnerability, even for functions containing up to thousands of lines. Even when developers know a function is vulnerable, they must spend time to locate the specific statements to edit. Wang *et al.* [20] propose GINN, an advanced GGNN model, and show its ability to localize null pointer dereferences, among several GINN applications. This work seems closest to ours but their research direction is orthogonal: we regard graph-based models in general, ignoring the subtle differences among variants; we instead study the complimentary effects between graph-based and sequence-based models, and leverage this distinction to better localize vulnerabilities. So for our proof of concept, we just pick the popular GGNN architecture as baseline.

VII. THREATS TO VALIDITY

Cross Validation. We tried to imitate a pragmatic real-world scenario, where we train the model on the past vulnerabilities and test on the latest samples. However, due to the small size of the D2A-TEST split, the model's reported results may not be generalizable. To minimize this threat, we did ten-fold cross-validation for all baseline models in the setting of Section V-B. Table VII reveals the same trend that VELVET-ENSEMBLE is winning by a clear margin on both metrics.

TABLE VII: Ten-fold cross-validation for Section V-B

	Vul-LOC Acc	Distance
VELVET-ENSEMBLE	**39.9%**	**8.6**
VELVET-GGNN	34.4%	9.8
VELVET-TRANSFORMER	37.9%	9.1
VELVET-TRANSSAND	39.3%	9.5
VELVET-GREAT	37.9%	8.7
VELVET-HOPPITY	35.5%	10.1

Cross-project. VELVET focuses on intra-project vulnerable patterns, since we expect the model to learn from the project's history and apply its knowledge to the project's new commits. To minimize the threats brought by such settings, we further study the model's cross-project performance under the Section V-B setting. We pick the D2A `Apache-HTTPD` project for evaluation and the other projects for training. The results show VELVET can achieve 41.8% Vul-LOC accuracy for top-1 and 60.0% for top-5, dropping a bit compared with the intra-project setting but still very promising.

Data Generalizability. Our results might be biased by the underlying data collection process, conducted by Zheng *et al.* [23]. However, collecting vulnerability data at statement granularity is hard. We have not found alternative real-world data with such fine-granular information to fully evaluate generalizability. To minimize this threat, we also evaluate VELVET on a synthetic dataset and show it performs better than static analysis alternatives.

VIII. CONCLUSION

We introduced VELVET, an ensemble model to efficiently capture local and global code context and understand vulnerable patterns at the individual statement level. Our evaluation showed that VELVET is effective for two vulnerability localization tasks on both synthetic and real-world data. Our designed workflow of pre-training on synthetic data and then fine-tuning on real-world data provides a practical solution to the real-world dataset scarcity problem.

ACKNOWLEDGMENT

This work is supported in part by NSF grants CCF-2107405, CCF-1845893, CCF-1815494, IIS-2040961, DARPA/NIWC-Pacific N66001-21-C-4018, and IBM. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect those of the US Government, NSF, DARPA, or IBM.

REFERENCES

[1] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 672–681, IEEE Press, 2013.

[2] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford, "Questions developers ask while diagnosing potential security vulnerabilities with static analysis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 248–259, ACM, 2015.

[3] B. Liu, L. Shi, Z. Cai, and M. Li, "Software vulnerability discovery techniques: A survey," in *2012 fourth international conference on multimedia information networking and security*, pp. 152–156, IEEE, 2012.

[4] S. Suneja, Y. Zheng, Y. Zhuang, J. Laredo, and A. Morari, "Learning to map source code to software vulnerability using code-as-a-graph," 2020.

[5] L. Buratti, S. Pujar, M. Bornea, S. McCarley, Y. Zheng, G. Rossiello, A. Morari, J. Laredo, V. Thost, Y. Zhuang, and G. Domeniconi, "Exploring software naturalness through neural language models," 2020.

[6] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pp. 757–762, 2018.

[7] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'2018)*, 2018.

[8] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," 2018.

[9] B. Li, K. Roundy, C. Gates, and Y. Vorobeychik, "Large-scale identification of malicious singleton files," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pp. 227–238, ACM, 2017.

[10] D. Maiorca and B. Biggio, "Digital investigation of pdf files: Unveiling traces of embedded malware," *IEEE Security & Privacy*, vol. 17, no. 1, pp. 63–71, 2019.

[11] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro, "Droidsieve: Fast and accurate classification of obfuscated android malware," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pp. 309–320, ACM, 2017.

[12] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems*, pp. 10197–10207, 2019.

[13] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?," 2020.

[14] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "Deepwukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, Apr. 2021.

[15] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations*, 2018.

[16] P. Yin, G. Neubig, M. Allamanis, M. Brockschmidt, and A. L. Gaunt, "Learning to represent edits," in *International Conference on Learning Representations*, 2019.

[17] V. J. Hellendoorn, C. Sutton, R. Singh, P. Maniatis, and D. Bieber, "Global relational models of source code," in *International Conference on Learning Representations*, 2020.

[18] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *International Conference on Learning Representations*, 2020.

[19] S. K. Dash, M. Allamanis, and E. T. Barr, "Refinym: Using names to refine types," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, (New York, NY, USA), p. 107–117, Association for Computing Machinery, 2018.

[20] Y. Wang, K. Wang, F. Gao, and L. Wang, "Learning semantic program embeddings with graph interval neural network," *Proc. ACM Program. Lang.*, vol. 4, Nov. 2020.

[21] J. Gao, Y. Jiang, Z. Liu, X. Yang, C. Wang, X. Jiao, Z. Yang, and J. Sun, "Semantic learning and emulation based cross-platform binary vulnerability seeker," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[22] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*, pp. 590–604, IEEE, 2014.

[23] 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 *Proceedings of the ACM/IEEE 43rd International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '21, (New York, NY, USA), Association for Computing Machinery, 2021.

[24] NIST, *Juliet test suite v1.3*, 2017. https://samate.nist.gov/SRD/testsuite.php.

[25] MITRE, *Common Weakness Enumeration*, 2020. https://cwe.mitre.org/data/index.html.

[26] *Github Repository for This Paper's data and Code*, 2021. https://github.com/ARiSE-Lab/VELVET.

[27] Y. Li, S. Wang, and T. N. Nguyen, "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*, ESEC/FSE 2021, (New York, NY, USA), p. 292–303, Association for Computing Machinery, 2021.

[28] D. Tarlow, S. Moitra, A. Rice, Z. Chen, P.-A. Manzagol, C. Sutton, and E. Aftandilian, "Learning to fix build errors with graph2diff neural networks," 2019.

[29] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," 2017.

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, p. 6000–6010, 2017.

[31] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," in *ACL*, pp. 4998–5007, 2020.

[32] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics*, 2021.

[33] T. Lutellier, V. H. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: Combining context-aware neural translation models using ensemble for program repair," 2020.

[34] Y. Ding, B. Ray, D. Premkumar, and V. J. Hellendoorn, "Patching as translation: the data and the metaphor," in *35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, 2020.

[35] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, (Online), pp. 1536–1547, Association for Computational Linguistics, Nov. 2020.

[36] V. Kovalenko, F. Palomba, and A. Bacchelli, "Mining file histories: Should we consider branches?," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 202–213, 2018.

[37] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?," in *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, (New York, NY, USA), p. 485–495, Association for Computing Machinery, 2009.

[38] joern.io, *Joern*, 2021. https://github.com/octopus-platform/joern.git.

[39] P. Fernandes, M. Allamanis, and M. Brockschmidt, "Structured Neural Summarization," in *International Conference on Learning Representations (ICLR)*, 2019.

[40] NIST, *National Vulnerability Database (NVD)*, 2020. https://nvd.nist.gov.

[41] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "Vuldeelocator: A deep learning-based fine-grained vulnerability detector," 2020.

[42] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang, "Leopard: Identifying vulnerable code for vulnerability assessment through program metrics," in *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, p. 60–71, IEEE Press, 2019.

[43] M. Jimenez, R. Rwemalika, M. Papadakis, F. Sarro, Y. Le Traon, and M. Harman, "The importance of accounting for real-world labelling when predicting software vulnerabilities," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, (New York, NY, USA), p. 695–705, Association for Computing Machinery, 2019.

[44] C. Calcagno and D. Distefano, "Infer: An automatic program verifier for memory safety of c programs," in *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, (Berlin, Heidelberg), p. 459–465, Springer-Verlag, 2011.
[45] D. A. Wheeler, *FlawFinder*. https://dwheeler.com/flawfinder/.
[46] S. S. Inc., *Rough Audit Tool for Security*, 2013. https://github.com/andrew-d/rough-auditing-tool-for-security.
[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* (K. Havelund, G. Holzmann, and R. Joshi, eds.), (Cham), pp. 3–11, Springer International Publishing, 2015.
[48] GitHub, *GitHub Code Security*, 2021. https://docs.github.com/en/code-security.
[49] Codacy, *Codacy Security Scan*, 2021. https://docs.codacy.com/repositories/security-monitor/.
[50] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," 2019.
[51] R. Abreu, P. Zoeteweij, and A. J. c. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pp. 39–46, 2006.
[52] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pp. 89–98, 2007.
[53] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, Aug. 2011.
[54] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, (New York, NY, USA), p. 273–282, Association for Computing Machinery, 2005.
[55] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pp. 153–162, 2014.
[56] M. Papadakis and Y. Le Traon, "Using mutants to locate "unknown" faults," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 691–700, 2012.
[57] L. Zhang, L. Zhang, and S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, (New York, NY, USA), p. 765–784, Association for Computing Machinery, 2013.
[58] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," ISSTA 2019, (New York, NY, USA), p. 169–180, Association for Computing Machinery, 2019.
[59] Z. Zhang, Y. Lei, X. Mao, and P. Li, "Cnn-fl: An effective approach for localizing faults using convolutional neural networks," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 445–455, 2019.
[60] Y. Lou, A. Ghanbari, X. Li, L. Zhang, H. Zhang, D. Hao, and L. Zhang, "Can automated program repair refine fault localization? a unified debugging approach," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, (New York, NY, USA), p. 75–87, Association for Computing Machinery, 2020.