Dataflow Analysis-Inspired Deep Learning for Efficient Vulnerability Detection

Benjamin Steenhoek
Iowa State University
Ames, Iowa, USA
benjis@iastate.edu

Hongyang Gao
Iowa State University
Ames, Iowa, USA
hygao@iastate.edu

Wei Le
Iowa State University
Ames, Iowa, USA
weile@iastate.edu

ABSTRACT

Deep learning-based vulnerability detection has shown great performance and, in some studies, outperformed static analysis tools. However, the highest-performing approaches use token-based transformer models, which are not the most efficient to capture code semantics required for vulnerability detection. Classical program analysis techniques such as dataflow analysis can detect many types of bugs based on their root causes. In this paper, we propose to combine such causal-based vulnerability detection algorithms with deep learning, aiming to achieve more efficient and effective vulnerability detection. Specifically, we designed DeepDFA, a dataflow analysis-inspired graph learning framework and an embedding technique that enables graph learning to simulate dataflow computation. We show that DeepDFA is both performant and efficient. DeepDFA outperformed all non-transformer baselines. It was trained in 9 minutes, 75x faster than the highest-performing baseline model. When using only 50+ vulnerable and several hundreds of total examples as training data, the model retained the same performance as 100% of the dataset. DeepDFA also generalized to real-world vulnerabilities in DbgBench; it detected 8.7 out of 17 vulnerabilities on average across folds and was able to distinguish between patched and buggy versions, while the highest-performing baseline models did not detect any vulnerabilities. By combining DeepDFA with a large language model, we surpassed the state-of-the-art vulnerability detection performance on the Big-Vul dataset with 96.46 F1 score, 97.82 precision, and 95.14 recall. Our replication package is located at https://doi.org/10.6084/m9.figshare.21225413.

ACM Reference Format:
Benjamin Steenhoek, Hongyang Gao, and Wei Le. 2024. Dataflow Analysis-Inspired Deep Learning for Efficient Vulnerability Detection. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3623345

1 INTRODUCTION

Software vulnerabilities cause great harm to people and corporations. Many Internet users have had their personal information breached because of security vulnerabilities, with common reports of breaches exposing millions of records [52]. The average data breach costs the target company $4.24 million, according to IBM's 2021 report [2]. The number of vulnerabilities is growing every year, as reported by the Common Vulnerability Enumeration (CVE) from 2016-2021 [1]. Due to its importance, we urgently need to develop effective and automatic vulnerability detection tools.

The rapid advance of AI technologies has motivated software companies to invest heavily in deep learning-based vulnerability detection tools [39, 55]. These tools have outperformed traditional static analysis [11, 20, 38]. Recently, large language models (LLMs) have reported state-of-the-art results; LineVul [24], a recent model based on CodeBERT, reported 91 F1 score on a commonly used real-world vulnerability dataset [22].

However, LLMs require large amounts of training data and computational resources for training and inference (see § 5.4), but a large volume of high-quality vulnerability detection data is hard to get. They also can fail to detect vulnerabilities beyond the training dataset (see § 5.5); for example, the top-performing transformer models LineVul and UniXcoder were not able to detect any of the real-world vulnerabilities in DbgBench [9]. Furthermore, by using solely text tokens, these models may not effectively learn program semantics, such as program values along paths, propagation of taint values, and security-sensitive API calls along the control flow paths. The performance of these models can be further improved when we consider such information (see §5.3).

In this paper, we explore the idea of combining *dataflow analysis (DFA)* algorithms with deep learning to develop small, efficient, yet effective models for vulnerability detection. In prior literature [16, 53], deep learning integrated with domain-specific knowledge and algorithms has reported improved performance and better generalization to unseen data, while using less data and computational resources.

Dataflow Analysis (DFA) computes the data usage patterns and relations in the control flow graph (CFG) of a program and reports a vulnerability based on its root cause, i.e., whether the values and data relations collected from the program indicate the occurrence of the vulnerable conditions. *Graph learning* (*learning based on graph neural networks (GNN)*) can aggregate and propagate information in the graph in a similar fashion to DFA. In this paper, we explore the analogy between DFA and the GNN message-passing mechanism and design an embedding technique that encodes dataflow information at each node of the CFG. Specifically, we leverage the efficient *bit-vector* representation of dataflow facts to encode the definitions and uses of the variables. Graph learning on such an embedding propagates and aggregates dataflow information and thus simulates the dataflow computation as done in DFA. Using this approach, we hope that the learned graph representation can better encode program semantic information, e.g., *reaching definitions*, which will be very useful for accurate vulnerability detection.

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

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3623345

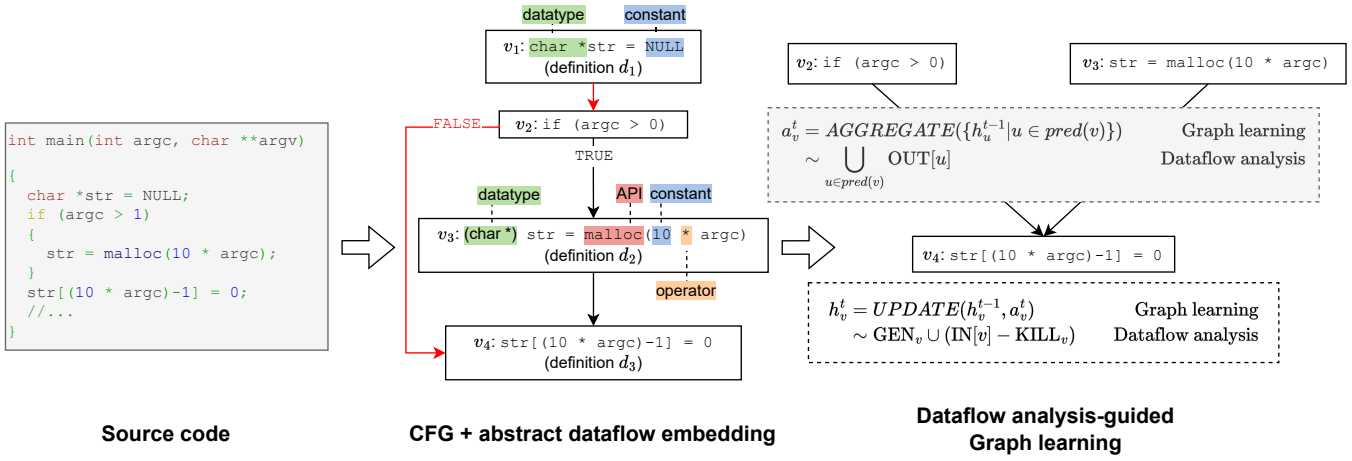

Figure 1: Overview of DeepDFA

Based on this rationale, we developed an *abstract dataflow embedding* that can map variable definitions of individual programs to a common space so that the model can compare and generalize data usage patterns (dataflow) related to vulnerabilities across programs. We selected a graph learning architecture whose aggregate and update functions worked most effectively for the dataflow propagation.

Our evaluation shows that DeepDFA is substantially faster than our baseline models in terms of both training and inference time. It only took 9 minutes to train, and inference on a CPU took 5.8 ms/example. This remarkable efficiency permits applications for personalized training and inference in non-GPU environments. It is also efficient in its use of training data, achieving its best F1 score using only 50+ vulnerable examples and several hundred total examples (§ 5.4). This frugality allows applications within a single development team, where it may be impractical to collect thousands of vulnerable examples. Yet, DeepDFA still outperformed all non-transformer baselines (§ 5.3) and retained its performance on unseen projects better than all baseline models (§ 5.5). Additionally, when applied to a real-world benchmark of unseen projects, DbgBench [9], DeepDFA detected 8.7 out of 17 of bugs (averaged over 3 runs) and correctly reported 3 out of 5 patched programs as non-vulnerable (§ 5.5). In comparison, the highest-performing baselines, LineVul [24] and UniXcoder [26], did not detect any vulnerabilities. We also show that DeepDFA's learned representation can be used with other models to further improve their performance. By combining UniXcoder with DeepDFA, we surpassed state-of-the-art performance with 96.46 F1 score, 97.82 precision, and 95.14 recall.

In summary, we made the following contributions in this paper:

(1) We designed an abstract dataflow embedding to enable deep learning to generalize semantics/dataflow patterns of vulnerabilities across programs (§4.1);
(2) We applied graph learning on the control flow graph (CFG) of the program and abstract dataflow embedding to simulate reaching definition dataflow analysis (§4.2);
(3) We implemented DeepDFA and experimentally demonstrated that DeepDFA outperforms baselines in vulnerability detection for effectiveness, efficiency, and generalization over unseen projects (§5);
(4) We provided rationale to help understand why DeepDFA performs well and is efficient (§3); and
(5) We surpassed the state-of-the-art vulnerability detection performance by combining DeepDFA and UniXcoder (§5).

2 OVERVIEW

We propose DeepDFA, a deep learning framework guided by dataflow analysis algorithms, shown in Figure 1. Given the source code of a potentially vulnerable program (left), we convert it to a CFG and encode the nodes using an *abstract dataflow embedding* which we designed. The CFG specifies the execution order of statements, and is the data structure on which dataflow analysis operates.

In the middle of the figure, we show our approach of computing abstract dataflow embeddings. In dataflow analysis, definitions of variables, e.g., `a=3`, are program specific. Applying to deep learning, we *abstract* these concrete definitions from different programs, and hypothesize that the usage patterns of the *abstract definitions* can be compared and summarized across programs during learning. To construct the abstract definitions, we used the properties of definitions that are important for vulnerability detection, based on domain knowledge from program analysis. Specifically, we considered the data types of the defined variable, the API calls, constants, and operators used to define the variables. Inspired by the bit-vector representation used in dataflow analysis, we encode the abstract definitions in a compact and very efficient fashion. We will provide more detailed design of this embedding in Section 4.1.

We used a bit-vector style of representing a *set* of abstraction definitions. This numerical representation can be directly used as the initial node representations for graph learning. In the right of the figure, we apply graph learning which *aggregates* the information from nodes like the "merge" operation performed in dataflow analysis, and also *updates* using the information at each nodes like

(a) Dataflow Analysis **(b) Graph Learning**

Figure 2: Analogy of information propagation in Dataflow Analysis and Graph Learning

the "update" operation performed in dataflow analysis. We provide more background on the analogy in Section 3.

Finally, we use the learned graph representation to classify whether the function is vulnerable or not. By directly propagating dataflow information through graph learning, we hope to present to the classifier a representation of the program which encodes useful information directly related to vulnerability, achieving efficient and effective vulnerability detection. The advantage of deep learning is that the mapping from the encodings of programs to the decisions are learned from the data, but in dataflow analysis, we need to manually craft rules to map from the dataflow analysis results to vulnerability decisions.

3 RATIONALE

In this section, we provide the relevant background of dataflow analysis for vulnerability detection and graph learning. It provides understanding on why our approach is efficient and effective. Then, we compared the closely related work that also considers dataflow in deep learning to clarify the novelty of our work.

3.1 Dataflow Analysis for Vulnerability Detection

Dataflow analysis (DFA) is a method for computing data usage patterns in a program. In addition to compiler optimization, dataflow analysis is an important method for vulnerability detection. One instance of dataflow analysis, called *reaching definition analysis*, reports at which program points a particular variable definition can *reach*. A definition *reaches* a node when there is a path in the CFG that connects the definition and the node, and the variable is not redefined along the path. The reaching definition analysis can detect a null-pointer dereference vulnerability based on its root cause when it identifies that a definition of an NULL pointer reaches a dereference of the pointer. Similarly, it is a causal step to detect many other vulnerabilities such as buffer overflows, integer overflow, uninitialized variables, double-free and use-after-free [12].

DFA uses two equations to propagate the dataflow information through the neighboring nodes in the CFG, namely *meet operator* and *transfer function* [4]. The meet operator aggregates the dataflow sets from its neighbors. The transfer function updates the dataflow set using the information available in the node v. In the reaching definition analysis, the dataflow set is a set of definitions that reach a program point. A simple approach of performing a DFA is the *Kildall method* [33]. It iteratively propagates the dataflow information to the neighbors of v in the CFG, one step at a time. The algorithm terminates when the dataflow information of all nodes stops changing, denoted a *fixpoint*. At termination, all nodes will incorporate the dataflow information from all other relevant nodes. When used for vulnerability detection, this information is compared to a user-specified vulnerability condition to determine whether a vulnerability has occurred in the program.

3.2 Analogy of Graph Learning and Dataflow Analysis

Graph learning starts with an initial node representation, and then it performs a fixed number of iterations of the message-passing algorithm [25] to propagate information through the graph. The initial node representation is generally a fixed-size continuous vector which represents the content of the node. At each iteration, each node aggregates information from its neighbors, and then updates its state to integrate the information. The two steps are done through the *AGGREGATE* and *UPDATE* functions, similar to the two dataflow equations of meet operator and transfer function. These functions can be simple numerical equations or neural networks. After iteration is done, all node representations are combined to produce a graph-level representation, which is passed to a classifier layer to make a prediction.

In Figure 2, we visualize the analogy between graph learning and dataflow analysis on a snippet of CFG. In the CFG, each node is a statement, and each edge indicates the order of execution between two statements. In Figure 2a, we show the two dataflow equations [4] that define a reaching definition dataflow analysis.

$$\text{meet operator:} \quad IN[v] = \bigcup_{u \in pred(v)} OUT[u] \tag{1}$$

$$\text{transfer function:} \quad OUT[v] = GEN_v \bigcup (IN[v] - KILL_v) \tag{2}$$

where $IN[v]$ and $OUT[v]$ are the sets of dataflow located at the beginning and end of a statement. GEN_v and $KILL_v$ represent the dataflow *generated* (new definitions) and *killed* (overwritten definition) in node v. Reaching definition is a *may* dataflow problem and thus the meet operator used *union* to merge the dataflow information from its predecessors. Meanwhile, reaching definition is a *forward* dataflow problem, and thus we used $IN[v]$, GEN_v, and $KILL_v$ to compute the dataflow at the exit of the statement.

In Figure 2b, we show an analogous behavior of graph learning.

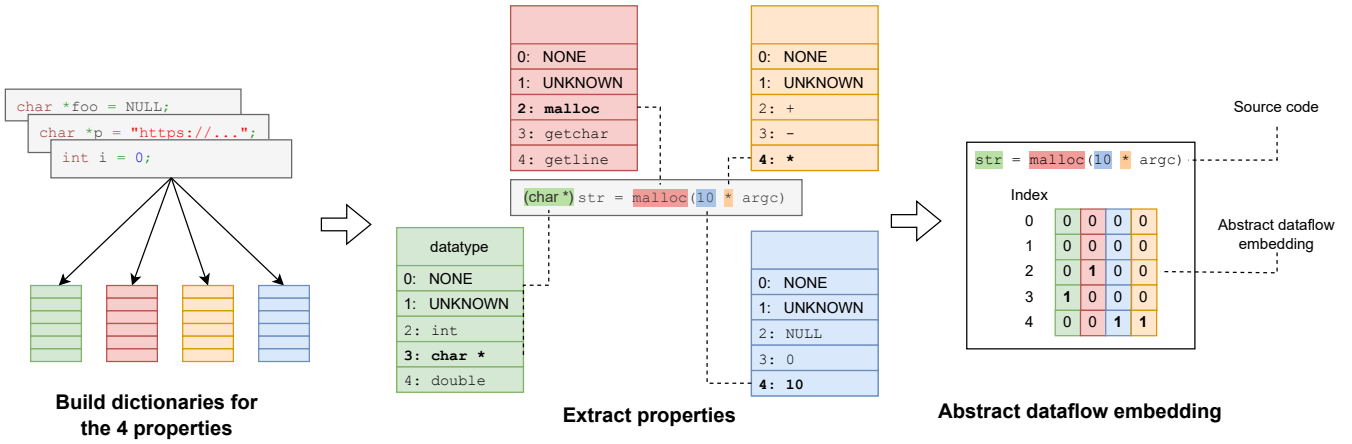

Figure 3: Abstract dataflow embedding generation

$$\text{aggregate:} \quad a_v^t = AGGREGATE(\{h_u^{t-1} | u \in pred(v)\}) \tag{3}$$

$$\text{update:} \quad h_v^t = UPDATE(h_v^{t-1}, a_v^t) \tag{4}$$

where a_v^t denotes the aggregated information from the neighboring nodes and h_v^t denotes the state of node v after t iterations of message-passing (analogous to $OUT[v]$). We set t as a hyperparameter.

3.3 The Novelty of Our Work

Previously, researchers have proposed to integrate dataflow information with deep learning for program analysis tasks. A category of approaches similar to Devign [56] used data dependency graphs as a part of the program representation on which deep learning is performed. However, Devign used word embeddings to encode statements into vector representations based on their unstructured text content. Such an encoding, even propagated through data dependency edges, cannot directly capture the dataflow patterns.

ProGraML [17] has developed graph learning on LLVM IR code and applied it for compiler optimization tasks. It is another work that pointed out the analogy between DFA and graph learning. Their solution is to modify CFGs by creating instruction nodes and data nodes separately. ProGraML adds control-flow edges between instruction nodes and data-flow edges between the data nodes. However, this work encoded nodes using an embedding which only represents LLVM IR operators and variable types. This approach is very coarse-grained in that many statements can have the same operators and variable types, but they will lead to different dataflow. Therefore, similar to Devign, the propagation of such an encoding even along dataflow edges does not directly capture dataflow patterns.

Our abstract dataflow embedding attempts to directly represent the variable definitions which are propagated in DFA and is modeled after the bit-vector representation used in DFA, which allows the network to learn the operations of the dataflow analysis algorithm. We also target a specific problem (reaching definitions, which was not targeted by ProGraML), for which the results of DFA are directly useful and pertinent to vulnerability detection (e.g. § 4.2).

4 APPROACH

Based on the analogous behaviors of DFA and GNN, we designed a node embedding that can represent the dataflow set at each node. We developed DeepDFA, a deep learning framework which conducts graph learning on the CFG of a program and propagates dataflow information for vulnerability detection.

4.1 Abstract Dataflow Embedding

In dataflow analysis, we use a *bit vector* to represent the dataflow set at each node. A bit vector consists of n bits of 0s and 1s. Its length is the size of the domain. A bit is set to 1 if its corresponding element is present in the set. In reaching definition analysis, the domain consists of all the definitions in the program, and the bits are set to "1" if the corresponding definitions reach the node. For example, in Figure 1, the program contains three definitions at nodes v_1, v_3, and v_4 so the reaching definition analysis uses a bit vector `[0 0 0]` to initialize each node at the beginning of the analysis. This bit vector represents $OUT[v]$ in the dataflow equations (See Section 3.2). It is updated at each step of propagation, and when the analysis terminates, the bit vectors for each node represent all possible definitions that can reach *that* node.

The bit-vector representation of reaching definition analysis efficiently encodes program semantic features related to vulnerability detection. The definitions of programs can be quickly obtained at the node via lightweight analysis locally at the statements. However, in graph learning, we cannot directly use the bit vector of definitions as the node embedding. This is because in dataflow analysis and the domain of definitions are both specific to a program. In other words, different programs have different variable definitions; the bit vectors of each program thus have different lengths and the elements (each definition) are not comparable either. Whereas, in graph learning, we want to extract dataflow patterns of vulnerabilities from all the programs in the training dataset. Thus, we need to have a "global" definition set that can be used to specify definitions for different programs, so that graph learning can compare them and generalize from them.

To address this challenge, we map all the concrete definitions in the programs in a training dataset to *abstract definitions* by identifying important properties of the definitions. Following a list of attack surfaces identified by Moshtari et al. [41], we designed the following four properties that can encompass the attack surfaces of a vulnerability and used them to represent a definition:

(1) API call: the call to library or system functions used to define a variable, e.g. `malloc` and `strlen`.
(2) Data type: the data type of the variable being assigned, e.g. `int`, `char*` and `float`.
(3) Constant: the constant values assigned in the definition, e.g. `NULL`, `-1` and the hard-coded string `"foo"`.
(4) Operator: the operators used to define a variable, e.g. `+`, `-` and `*`.

We analyze a large corpus of programs, e.g., the training set, and collect the top-k frequently used API calls, data types, constants and operators to construct a dictionary. k is a hyperparameter of DeepDFA. We select only the top-k keys because the representations of user-defined names of APIs and data types cannot be generalized across programs unless they are represented frequently in the dataset.

In Figure 3, we show an example of abstract dataflow embedding for an example d_2 in Figure 1: `str = malloc(10 * argc)`. This definition used an API call, `malloc`, with the constant `10`, operator `*`, and data type `char*`. Contrasted with the 3-bit bit vector (the example in Figure 1 includes three variable definitions) that represents a concrete definition in dataflow analysis for this program, the abstract embedding is larger but a fixed size, consisting of 5x4 elements for this example. Here, 4 is the four properties we considered and 5 is the hyper-parameter k we mentioned above, which defines the size of the pre-defined dictionary, and the length 5 hot-vector encoding represents the value of the property. Because the vector that encodes the abstract dataflow embedding has a fixed size, our embedding approach can scale to any program size in the dataset without impacting the model's efficiency. The vectors in different programs encode common properties of definitions, so the model can capture the dataflow patterns across programs.

Abstraction potentially brings in approximation. Using the abstract dataflow embedding, two different definitions may lead to the same encoding. The embedding is designed to be sparse enough that within a program, unique definitions are often represented by unique embedding keys, which allows the model to distinguish definitions within the same function, similar to the bit-vector used in dataflow analysis.

4.2 Using Graph Learning to Propagate Dataflow Information

Our goal in utilizing graph learning is to learn a node embedding that contains dataflow information. Without loss of generality, we use *reaching definition* as an instance of dataflow analysis for our explanations. Our approach takes the following steps. First, we construct the CFG for a program. Second, we perform static analysis to identify all the definitions in the CFG. We then initialize each node of the CFG using the abstract dataflow embedding, based on whether the node is a definition or not. The abstract dataflow embedding is computed from all the programs in the training dataset (see §4.1 for details).

Once the nodes are initialized, we apply the message-passing algorithm [25] from graph learning to propagate the dataflow information throughout the CFG, similar to *Kildall's method* [33]. The main differences are that (1) we propagate the abstract dataflow embeddings of the CFG nodes, and (2) instead of using the dataflow equations of transfer function and meet operator, we alternatively apply the *AGGREGATE* and *UPDATE* functions defined in Equations 3 and 4 (See §3.3). Although the analogy applies for all GNN architectures trained with message-passing, we implemented our approach using a *Gated Graph Sequence Neural Network (GGNN)* [35], where *AGGREGATE* is an *Multi-Layer Perceptron (MLP)* and *UPDATE* is a *Gated Recurrent Unit (GRU)*; we will use this architecture as an example to compare the two algorithms.

When dataflow information arrives at the merge point of a branch in CFG, graph learning applies the *AGGREGATE* function. Specifically, in GGNN, the MLP calculates a weighted sum of the representations of multiple neighboring predecessors, resulting in a single vector; this fulfills the same function as the meet operator. When dataflow information arrives at a new node, the *UPDATE* function in graph learning computes the next state by combining the information in the current node with the output of *AGGREGATE* from its predecessors. Specifically, in GGNN, the GRU selectively forgets portions of the previous state and integrates new information from the current node and from the neighboring states, similar to the set union/difference with GEN/KILL performed in the transfer function. Through applying *AGGREGATE* and *UPDATE*, the initial embedding will be updated with the dataflow information from the neighboring nodes, similar to the effect of dataflow analysis.

As Cummins et al. [17] noted, DFA iterates to a fixpoint and thus propagates information throughout the entire graph, while graph learning performs a fixed number of iterations t and thus propagates to neighbors in a distance t. We set t to the setting which maximized validation-set performance.

Finally, we combine the learned abstract node embeddings to produce graph level representation using *Global Attention Pooling* [35], and pass it to a classifier to predict the function as vulnerable or non-vulnerable.

The *AGGREGATE* and *UPDATE* functions are learned from labeled data during training, rather than using a fixed formula as in dataflow analysis. By learning from data, we provide an alternative solution to the challenges that often block dataflow analysis such as tracking pointers and handling library calls. Importantly, we no longer need to explicitly specify vulnerability conditions, as required in static analysis. Through learning from training examples, the classifier can capture patterns of dataflow information that represent various types of vulnerabilities and also select the relevant dataflow information for vulnerability detection.

In Table 1, we step through a reaching definition analysis for the CFG example in Figure 1 to demonstrate how dataflow information propagates through the graph and how our approach uses dataflow information for vulnerability detection.

The row *Iteration 0* shows the initialization of each node in the reaching definition analysis. At iteration 1, the DFA updates

Table 1: $OUT[v]$ at each iteration of DFA

Iteration	v_1	v_2	v_3	v_4
0	[0 0 0]	[0 0 0]	[0 0 0]	[0 0 0]
1	[1 0 0]	[0 0 0]	[0 1 0]	[0 0 1]
2	[1 0 0]	[1 0 0]	[0 1 0]	[0 1 1]
3	[1 0 0]	[1 0 0]	[0 1 0]	[1 1 1]

$OUT[v_1]$, $OUT[v_3]$ and $OUT[v_4]$ using the transfer function to indicate that the new definitions are introduced at the nodes. At iteration 2, $OUT[v_1]$ (including d_1) propagates to v_2 and $OUT[v_3]$ (including d_2) propagates to v_4, through the CFG edges. At iteration 3, the meet operator is used to combine $OUT[v_2]$ and $OUT[v_3]$. Specifically, $IN[v_4] = \bigcup\{OUT[v_2], OUT[v_3]\}$, computed as $[1\,0\,0] \vee [0\,1\,0] = [1\,1\,0]$; then the transfer function combines $IN[v_4]$ with GEN_{v_4}, resulting in $OUT[v_4] = [1\,1\,1]$.

After the DFA algorithm terminates, the final states of the nodes are used to detect vulnerabilities. The state of v_4 is $[1\,1\,1]$, which indicates that both d_1 and d_2 may reach v_4 depending on the program values. Because the definition d_1 `: str = NULL` can reach the dereference at v_4, we can conclude that this program has a null-pointer dereference vulnerability. Similarly, in graph learning, after a fixed number of iterations, all the node representations are combined using a graph readout operation to produce a graph-level representation, which is used to predict for vulnerability detection. Programs with the null-pointer dereference bugs will have the same abstract definitions characterized by the `char*` type and the constant `NULL` to reach the pointer dereference statements. We believe that the dataflow information represented by DeepDFA will allow a relatively simple classifier to recognize this pattern among the training dataset.

5 EVALUATION

In the evaluation, we studied 3 research questions:

(1) Is DeepDFA **effective** for finding vulnerabilities?
(2) Is DeepDFA **efficient**, both in terms of training data and computational resources?
(3) Can DeepDFA **generalize** to unseen projects?

We also performed ablations on DeepDFA to understand the effects of each feature on its performance.

5.1 Implementation

To explore whether DeepDFA can advance the state-of-the-art, we created two settings, **DeepDFA** and **DeepDFA+LLM**. We implemented DeepDFA using the GGNN architecture [35] and based on LineVD's implementation[1], using PyTorch and DGL[2]. We used Joern[3] to parse the CFGs because it does not require compilation; this allows our approach to be utilized out-of-the-box given only the source code, without extra configuration. To implement DeepDFA+LLM, during training and inference, we combine the graph embedding generated by DeepDFA's graph readout stage with the sentence embedding produced by the final self-attention layer of LLM. The embeddings are concatenated and fed into a feed-forward classifier layer; both embeddings and the classifier are trained jointly. We believe that providing dataflow information can improve the LLM embedding, as LLM is trained exclusively from text and it is hard to learn dataflow relations among all the dependencies of tokens.

[1]https://github.com/davidhin/linevd
[2]All of our code and data are available at https://doi.org/10.6084/m9.figshare.21225413
[3]Joern version 1.1.1072, available at https://joern.io

Table 2: Hyperparameters used for training DeepDFA.

Hyperparameter	Value
λ (learning rate)	$1e^{-3}$
L_2 weight	$1e^{-2}$
k (threshold)	1000
t (number of GNN steps)	5
Hidden size	32
# output layers	3
Batch size	256

To avoid data leakage, we extracted the initial abstract dataflow embedding from the training set only. We set the hyperparameters k and t based on the best validation performance through our experimentation. When $k = 1000$, the model covered most (79.38%) of the definitions in the test dataset. That means, the dictionary still misses some APIs, constants, data types or operators that occur in the test data set but are not frequent in the training dataset. To learn a more general representation and improve the coverage for test dataset, in future work, we can train the abstract dataflow embedding using a very large dataset of code, e.g., using self-supervised learning without the need of vulnerability labels.

For reproducibility, Table 2 documents the hyperparameters we used for training DeepDFA.

5.2 Experimental setup

In the recent literature [13, 24, 36], vulnerability detection models are typically evaluated with the Devign [56] or Big-Vul [22] datasets, both of which contain real-world open-source C/C++ projects. In our evaluation, we used the Big-Vul dataset because (1) it is bigger than Devign, consisting of 188,636 functions with 10,900 (6%) vulnerable labels and 177,736 (94%) non-vulnerable labels, and (2) it reflects the imbalanced distribution of real-world code (Devign is a balanced dataset), with the minority of code being labeled as vulnerable [13]. To corroborate our results, we also evaluated the models on DbgBench [9], explained further in RQ3.

Scope: Currently, DeepDFA reports whether a function is vulnerable or not. We can further apply deep learning explanation tools to report line level vulnerabilities, as done in Li et al.'s work [36]. We leave this evaluation for future work. We evaluated DeepDFA on C/C++ programs, as done by the most deep learning-based vulnerability detection tools. However, we believe that DeepDFA can also be applied to other popular programming languages, such as Python and Java. This is because we extract the abstract dataflow embedding (API, datatype, literal, operator) from the training dataset, independent of the programming language.

RQ1: To evaluate the models' performance, we trained the models on the train/validation/test splits of 80/10/10% published by

the LineVul paper [24]. To address class imbalance while training DeepDFA, we undersampled the majority class (non-vulnerable) following Japkowicz et al. [31]; our initial studies found that this improved our performance on the validation set. We kept the original ratio of vulnerable/non-vulnerable labels for the validation and test sets. We used Joern[4] [54] to parse the code into its CFG representation. Joern could not parse some programs in the dataset (0.8%; see Appendix B for details), so we used the remaining data in our experiments. The performance of the baseline models was similar to the full dataset (see Appendices C & F for details).

We report the following performance metrics:

- **Precision** reports the portion of positive predictions which were correct: $P = \frac{TP}{TP+FP}$.
- **Recall** calculates the portion of positive examples which were recalled correctly: $R = \frac{TP}{TP+FN}$.
- **F1** is the harmonic mean between Precision and Recall: $F1 = 2 * \frac{P*R}{P+R}$. We used $F1$ to decide the highest performing model because it balances precision and recall, which are both important in an imbalanced dataset.

Since the model performance can vary with different random seeds [48], we trained the models 3 times with different random seeds and reported the mean score and standard deviation for each metric. We used McNemar's statistical test, following best practices [18], to confirm that our improvement is statistically significant, using the implementation in *statsmodels* v0.14.0 [47].

RQ2: To evaluate the models' efficiency in terms of computational resources, we measured the runtime and memory usage. These are often contested resources in deep learning workloads [30]. We report the following metrics for runtime and memory usage:

- **Training time**: the wall-clock time to execute one training run with one validation run per epoch
- **Inference time**: the average wall-clock time to predict for one example.
- **MACs**: the average number of Multiply-Accumulate operations[5] to predict for one example; this measures the performance independently of the computing platform [30, 49].
- **Parameter count**: the number of trainable parameters in the neural network model.

We evaluated the runtime on an AMD Ryzen 5 1600 3.2 GHz processor with 48GB of RAM and an Nvidia 3090 GPU with 24GB of GPU memory.

To evaluate the models' efficiency in terms of training data, we trained the models on progressively smaller subsets which we randomly sampled from Big-Vul (100%, 10%, 1%, 0.5%, 0.1%, shown in the columns of Table 6). Each subset includes the smaller subsets (e.g. 10% subset includes the 1% subset and 1% includes 0.5%). We generated 3 versions of the subsets using different random seeds and reported the mean and standard deviation F1 score. The goal of this study is to discover what are the minimum training data needed for these models to perform well on the test dataset.

RQ3: We prepared two experiments for this RQ. In the first experiment, we created a dataset from Big-Vul to evaluate how well the models generalize to unseen projects. This dataset consists of the *mixed-project* and *cross-project* two settings. To set up the mixed-project setting, we held out 10k randomly selected examples for the validation and test sets and used the rest for training, similar to the original method of partitioning the dataset. The training set and test set can and often do contain examples from the same project, though individual examples will not be duplicated between the two sets. To set up the cross-project setting, we held out 10k examples from randomly selected projects in Big-Vul for the validation and test sets, and used the rest of the projects for training. The projects in the test set are distinct from the projects in the training set. To mitigate the potential bias caused by the selection of projects, we repeated this process 5 times with different selections of the cross-project data and report the results of 5-fold cross validation.

In order to further evaluate generalization to unseen projects, we applied DeepDFA on buggy and patched programs from DbgBench [9]. DbgBench consists of a set of real-world C programs with bugs, which were analyzed and fixed by professional software engineers. The DbgBench programs are distinct from the programs in the Big-Vul dataset. We labeled the buggy functions using the fault locations documented in DbgBench; these were labeled by the consensus of multiple developers and were manually checked for correctness, and thus are more reliable than Big-Vul's labeling process based on bug-fixing commits. We included all functions which had a bug location marked as "buggy" and their corresponding patched versions in our study. We excluded the bugs marked as "Functional" because these bugs cannot be detected without program-specific bug constraints. We only included the patched versions which were modified by the developers' fixes, taking the first correct[6] developer patch which could be applied to the program. We included only one patch to reduce the effects of code duplication, which can unfairly bias test performance [5]. Since the models only view the function-level context, they will not produce a different prediction on the functions which were not modified by the patch. We also excluded 8 examples which could not be processed by Joern in order to fairly compare the models' performance scores. This resulted in a dataset of 22 programs: 17 buggy + 5 patched. We evaluated the checkpoints trained from 3 random seeds in Section 5.3 and report the mean performance scores.

Ablation study: We ran two ablation settings for each of the four abstract dataflow embedding features, resulting in eight settings: (1) using one feature at a time and (2) using three features at a time (leaving one out). In each setting, we trained DeepDFA on the Big-Vul [22] training dataset, and then evaluated on the Big-Vul test dataset and DbgBench [9].

Baselines: We compared against 7 non-transformer models: VulDeePecker, SySeVR, Draper, Devign, ReVeal, ReGVD, IVDetect, and 4 large language models: CodeBERT, LineVul, UniXcoder, and CodeT5 [7]. These models were developed recently with diverse architectures, and they represent the state-of-the-art of vulnerability detection models [48]. See Section 7 for an overview of the models and Appendix A in the supplementary materials for the details of our reproductions.

[4] https://joern.io

[5] We used DeepSpeed profiler to measure MACs [44]. https://www.deepspeed.ai

[6] Marked in DbgBench as "Developer fix" or "Different but Correct Fix", e.g. https://github.com/dbgbench/dbgbench.github.io/blob/master/patches/find.dbcb10e9/README.md

[7] We could not reproduce LineVD and ContraFlow on Big-Vul for function-level vulnerability detection.

Table 3: DeepDFA outperformed the baselines and can be used to further improve the existing model performance. All scores are reported as *Mean (Standard deviation)*. Note that VulDeePecker, SySeVR, Draper, and IVDetect performance were directly taken from the IVDetect paper [36], so we do not report the variance.

(a) Comparison with non-transformer models.

Model	F1	Precision	Recall
VulDeePecker	12.00	49.00	19.00
SySeVR	15.00	**74.00**	27.00
Draper	16.00	48.00	24.00
ReGVD	19.15 (2.65)	63.67 (4.43)	11.33 (1.94)
IVDetect	23.00	72.00	35.00
Devign	26.85 (0.97)	29.00 (0.38)	25.03 (1.67)
ReVeal	32.94 (0.75)	34.27 (1.58)	31.73 (0.65)
DeepDFA	**68.26** (0.16)	53.98 (0.06)	**92.81** (0.40)

(b) Comparison with transformer models.

Model	F1	Precision	Recall
CodeBERT	21.04 (6.72)	68.48 (11.76)	12.91 (5.51)
CodeT5	45.61 (0.71)	56.47 (6.22)	38.56 (2.45)
LineVul	93.23 (0.31)	97.32 (0.66)	89.48 (0.42)
UniXcoder	95.11 (0.21)	96.96 (1.14)	93.34 (1.23)
DeepDFA	68.26 (0.16)	53.98 (0.06)	92.81 (0.40)
DeepDFA+CodeT5	81.39 (0.96)	94.23 (2.98)	71.67 (1.09)
DeepDFA+LineVul	96.40 (0.13)	**98.69** (0.28)	94.22 (0.46)
DeepDFA+UniXcoder	**96.46** (0.09)	97.82 (0.99)	**95.14** (1.09)

5.3 Effectiveness

Comparison with non-transformer models: In Table 3a, we show that DeepDFA performed much better than the baseline models on F1 score and recall. DeepDFA's score was 47.51 higher than the average F1 score computed over all the baselines. In addition, compared to the other 6 models, DeepDFA reported lower variances for all the three metrics. This indicates that DeepDFA was more robust to random noise throughout training, and thus more likely to perform as expected after training.

Table 4: Results of statistical tests for model comparison.

Models compared	χ^2 statistic	p-value
LineVul vs. DeepDFA+LineVul	20.0	2.77×10^{-7}
UniXcoder vs. DeepDFA+UniXcoder	25.0	5.35×10^{-4}

The results show that our abstract dataflow embedding indeed encodes useful information for vulnerability detection, despite the fact that the node representation is small and the graph is simple. It is more effective than *property graphs* (a combination of AST, CFG, and PDG) used in Devign and Reveal. These baseline models represented nodes using unsupervised word embeddings [34, 40, 43], which do not have a direct relationship with vulnerabilities. In contrast, DeepDFA's node representation encodes the dataflow sets of reaching definitions, related to the root causes of vulnerabilities.

Comparison with transformer models: We compared DeepDFA with CodeBERT, LineVul, UniXcoder, and CodeT5 – the state-of-the-art among the transformer language models we evaluated. (see Appendix C for the performances of all the baseline models). Table 3b shows that DeepDFA performed considerably better than CodeBERT and CodeT5 in F1 score and had the smallest variance (among all the models) between runs.

Although UniXcoder and LineVul performed better than DeepDFA in terms of F1 score, DeepDFA's embedding can be combined with UniXcoder and LineVul to further improve their performance. We achieved state-of-the-art performance on all three metrics by adding DeepDFA's embedding to UniXcoder, with an F1 score of 96.46 (1.35 improvement), a Precision score of 97.82 (0.86 improvement), and a Recall score of 95.14 (1.80 improvement). Adding DeepDFA's embedding improved CodeT5 considerably – by 35.78 F1 score – and improved LineVul by 3.17 F1 score. We used McNemar's significance test, as recommended by Dietterich et al. [18], to confirm that the differences in performance were statistically significant ($p < 0.05$; see Table 4).

DeepDFA does not use any text-/token-level information such as variable and function names, yet it has achieved excellent performance. We believe that leveraging the domain-specific algorithm of reaching definition analysis to guide graph learning indeed plays an important role and that the embedding indeed encodes semantic features (e.g., data relations) that are important for vulnerability detection. The fact that DeepDFA can further improve the top-performing LLMs indicates that LLMs, which exclusively leverage text information, may not sufficiently learn the dataflow of code; DeepDFA thus provides the complementary information for vulnerability detection. We further believe that the examples which DeepDFA predicted incorrectly could be attributed to the fact that reaching definition analysis cannot handle all types of vulnerabilities. Thus, by adding other dataflow analyses such as live variable analysis, DeepDFA could further improve its performance. We will leave such an investigation to our future work.

Table 5: DeepDFA's training/inference time was faster than the baselines.

Model	Train time (ms)	Inference cost per example		
		GPU (ms)	CPU (ms)	MACs
LineVul	10h19m	11.1	1068.2	48.32 B
DDFA+LV	10h40m	15.4	1571.5	48.32 B
UniXcoder	11h16m	9.5	486.0	48.32 B
DDFA+UXC	13h22m	13.3	922.1	48.32 B
DeepDFA	**9m**	**4.6**	**5.8**	**40.27 M**

RQ1 result: DeepDFA performed much better than all non-transformer baselines. When combined with transformer models, it achieved the highest SOTA score on all metrics.

5.4 Efficiency

Efficiency of computational resources: In Table 5, we present the runtime comparison of DeepDFA, LineVul, and UniXcoder. Here, we did not list other models because their performances are much worse (shown in Table 3), and they took hours to train (see Appendix D in the supplementary material), compared to DeepDFA which finished training in 9 minutes (excluding data preprocessing time). In Appendix E, we also listed the sizes of the models in terms of the number of parameters.

Compared to UniXcoder, DeepDFA took 75x less time to train, 2x faster inference on GPU, and 84x faster inference on CPU. DeepDFA had the least parameters of all models, equal to 67% of the smallest model (ReVeal) and 0.3% of the highest-performing baseline model (UniXcoder). These results consistently indicate that DeepDFA excels in its efficiency compared to other models. This is possible because DeepDFA is based on the dataflow analysis's compact representation – bitvector, which captures the relevant semantic information in bits and thus is more efficient compared to tokenized strings. DeepDFA propagated information along only the domain-specific CFG edges, rather than associating every pair of tokens in an exhaustive fashion.

DeepDFA's short inference time due to a low number of MAC operations enables its use in non-GPU environments (which are common for software development) where large language models may not be easily deployed. DeepDFA's short training time enables techniques like per-project fine-tuning and hyperparameter tuning, which would be much more costly with the LLMs' training times of over 10 hours. Because of DeepDFA's small parameter count, it is ideal for resource-limited computing platforms such as mobile devices, where large models cannot be used [30].

Efficiency on training data: In Table 6, we report the performance of DeepDFA over reduced training dataset sizes, compared to the SOTA models, LineVul and UniXcoder. The columns "# data" and "# vul" list the number of training examples in each subset and, of these, the number of vulnerable examples. The results show that DeepDFA maintained a stable performance across small dataset sizes, even with only 0.1% of the training dataset, using only 151 training examples. In contrast, LineVul and UniXcoder steadily dropped in performance as the size of the training dataset decreased. At 0.1% data, LineVul's mean performance was only 29.75 in F1 and UniXcoder's was 4.36.

Table 6: DeepDFA retained its performance on limited data.

Portion	# data	# vul	F1		
			LineVul	UniXcoder	DeepDFA
0.1%	151	11	29.75	4.36	**55.24**
0.5%	755	56	77.69	**79.37**	68.17
1.0%	1,510	90	**84.62**	79.60	68.40
10.0%	15,091	885	86.67	**92.53**	68.44
100.0%	150,908	8,736	93.23	**95.11**	68.26

For project-specific training in applications within a single development team, a model which can learn efficiently from a small dataset is useful.

We believe that our model's stable performance over the reduced dataset and good performance with very small training datasets demonstrate the advantage of the small models and the effectiveness of domain-specific algorithms to guide model learning. DeepDFA is less prone to overfitting to datasets of limited size since it has fewer parameters than LineVul [8]. On the other hand, the transformer models require a large corpus of programs to learn the patterns among the unstructured token data.

RQ2 result: DeepDFA was considerably faster than the baselines; it took 9 minutes to train, 4.64 milliseconds for inference on GPU, and 5.8 milliseconds for inference on CPU. DeepDFA retained stable performance as the training dataset size was reduced. In a low-data scenario, DeepDFA outperformed LineVul and UniXcoder by 25.49 and 50.88 points F1 score.

5.5 Generalization

Cross-project evaluation on Big-Vul: We compared the models' F1 scores on the *cross-project* (shown as Cross F1) and *mixed-project* (shown as Mixed F1) settings to evaluate the models' capabilities of generalizing over unseen projects. Table 7 presents the highest-performing baseline models, LineVul and UniXcoder, compared to DeepDFA (the results of the other baseline models are available in

Table 7: How do the models handle unseen projects? Note the performance drop (ΔF1) from the cross-project to mixed-project setting.

Model	Mixed F1	Cross F1	ΔF1
LineVul	84.03	71.37	-12.66
UniXcoder	86.30	76.72	-9.58
DeepDFA	70.49	68.58	**-1.91**
DeepDFA+LineVul	87.89	71.88	-16.02
DeepDFA+UniXcoder	**89.85**	**78.07**	-11.77

the supplementary material, Appendix F). Among the most important metrics, ΔF1 shows how performance changes when the model is applied to unseen projects. Shown under Column ΔF1, DeepDFA only dropped 1.91 (2.7%) F1 score, compared to 12.66 (15.1%) drop for LineVul and 9.58 (11.1%) drop for UniXcoder. DeepDFA+UniXcoder reported the best performance for both the mixed-project and cross-project settings, improving on UniXcoder's mean F1 score by 3.55 and 1.35 points respectively; DeepDFA+LineVul also improved LineVul's F1 score in both settings.

Applying to DbgBench:

In Table 8, we report our experience of applying deep learning tools to real-world bug benchmarks. DeepDFA detected 8.7 out of 17 total bugs on average across 3 runs. DeepDFA also correctly predicted non-vulnerable for 3 out of 5 patched programs. On the other hand, neither of the competing LLMs, LineVul and UniXcoder, detected any bugs and in fact both models reported all programs as non-vulnerable with high confidence. This implies that these models were heavily biased to predict all examples in DbgBench as non-vulnerable. With the addition of DeepDFA, DeepDFA+LineVul and DeepDFA+UniXcoder's generalization greatly improved, yet they did not perform as well overall as DeepDFA alone. It should be noted that the bugs in DbgBench are very complex and took human experts hours to diagnose [9]. In the past, we have tried a variety of static analysis tools, such as Cppcheck[8] and Polyspace[9], to detect bugs in DbgBench, but we have not detected any of these bugs.

We believe that DeepDFA generalizes better because it does not rely on spurious features that may exist at token and text level, such as variable names and function names, as reported by previous research [13]. These spurious features are no longer correlated with vulnerabilities in unseen projects, as their input tokens will likely change. Our abstract dataflow embedding encodes the usage patterns of commonly used API calls, operators, constants, and data types. Such patterns can be extracted from unseen text and are directly related to the cause of the vulnerabilities, and thus might help DeepDFA generalize better over unseen projects.

> **RQ3 result:** DeepDFA had the smallest drop in F1 score (Δ F1) when applying to the vulnerabilities in the projects that are not seen in training datasets. DeepDFA was able to detect complex bugs in DbgBench and was able to distinguish the buggy and patched versions. The SOTA models, LineVul and UniXcoder, did not detect any bugs in DbgBench.

5.6 Ablation studies

Table 9 shows the model's performance on DbgBench. The model detected the most bugs when using all four features compared to other ablation settings. When using only one feature at a time, the model consistently missed 1-2 bugs which were detected by DeepDFA. When using three features at a time (leaving one out), the model still consistently failed to detect 1 bug which was detected by DeepDFA.

[8]https://cppcheck.sourceforge.io/
[9]https://www.mathworks.com/products/polyspace.html

Table 8: DeepDFA generalized to real-world bugs in DbgBench. Results are averaged over checkpoints from 3 random seeds. Buggy/Patched columns show the number of correct predictions on buggy/patched programs respectively.

Model	Buggy	Patched	Accuracy	F1
LineVul	0.0	**5.0**	22.73	0.00
DeepDFA+LineVul	**9.0**	1.3	46.97	60.67
UniXcoder	0.0	5.0	22.73	0.00
DeepDFA+UniXcoder	**9.0**	1.3	46.97	60.67
DeepDFA	8.7	3.0	**53.03**	**64.29**
Total	17.0	5.0	-	-

Table 9: Ablation study evaluated on DbgBench

Feature set	Buggy	Patched	Acc	F1
DeepDFA	**8.7**	**3.0**	**53.03**	**64.29**
API only	7.7	3.0	48.48	57.50
Datatype only	8.0	3.0	50.00	59.26
Literal only	7.7	3.0	48.48	57.50
Operator only	7.7	3.0	48.48	57.50
Api+datatype+literal	8.0	3.0	50.00	59.26
Api+datatype+operator	8.0	3.0	50.00	59.26
Api+literal+operator	8.0	3.0	50.00	59.26
Datatype+literal+operator	8.0	3.0	50.00	59.26
Total	17.0	5.0	-	-

Table 10 shows the model's performance on the Big-Vul test dataset. DeepDFA (integrating all the four features) performed the best out of all configurations. When testing one feature at a time, datatype by itself performed better than the other 3 features alone. When we used the combined feature sets, the model performed better than using only one feature.

6 THREATS TO VALIDITY AND DISCUSSIONS

Threats: We evaluated performance primarily on the Big-Vul dataset because this dataset was supported by all the baseline models. Compared to the Devign dataset, Big-Vul is imbalanced and can better reflect a real-world vulnerability detection scenario. However, Big-Vul's data collection process based on bug-fixing commits can introduce label noise and selection bias and as a result, the evaluation could fail to represent real-world performance. To address the selection bias, we studied settings which reflect more realistic scenarios with reduced training datasets and cross-project generalization; to address the label noise, we evaluated the models on additional real-world bugs collected in DbgBench, which were labeled by developers and manually checked.

The performances reported in RQ1, RQ2, and RQ3 will be affected by the random noise in the model training and, for RQ2, dataset selection. To mitigate this effect, we generated 3 versions of the subsets using different random seeds and reported the mean performance.

Table 10: Ablation study evaluated on the Big-Vul test dataset

Feature set	F1	Precision	Recall
DeepDFA	**68.26** (0.16)	53.98 (0.06)	**92.81** (0.40)
Datatype only	68.04 (0.19)	53.83 (0.31)	92.46 (0.30)
Literal only	62.50 (0.81)	50.52 (1.83)	82.46 (7.26)
Operator only	64.47 (0.33)	52.82 (1.59)	82.98 (4.90)
API only	63.67 (0.45)	50.66 (2.36)	86.14 (5.58)
API + datatype + literal	68.18 (0.10)	**54.06** (0.13)	92.28 (0.15)
API + datatype + operator	68.16 (0.13)	54.03 (0.18)	92.28 (0.15)
API + literal + operator	68.11 (0.14)	53.98 (0.20)	92.28 (0.15)
Datatype + literal + operator	68.12 (0.20)	54.04 (0.11)	92.11 (0.46)

The mixed-project and cross-project performance reported in RQ3 will be affected by the random selection of projects in the training/held-out datasets. To mitigate this effect, we performed 5-fold cross-validation and reported the mean performance.

Discussions: We believe our approach can be extended to *bit-vector* dataflow problems [29, 45]. All problems in this category contain a finite set of dataflow facts and have the same form of transfer functions and meet operators (see Equations 1 and 2). For example, live variables and available expressions [45] are bit-vector problems that are important for vulnerability detection [12]. We believe a new dataflow analysis can be integrated by: (1) defining an abstract dataflow embedding which can capture the dataflow set of the analysis, (2) configuring the neural network used as the aggregate function in GGNN to better simulate the meet operator (based on whether it is a union or intersection operation), and (3) reversing the CFG edges for backward dataflow problems (as reaching definition is a forward dataflow problem).

7 RELATED WORK

Many works have used GNN for vulnerability detection [6, 10, 14, 19, 21, 42, 51]. In several recent approaches, Devign [56], ReVeal [13], IVDetect [36], and LineVD [28] used GNN on program graph representations such as AST, CFG, and PDG, and annotated the nodes with unsupervised or pretrained word embeddings. The novelty of our work is a bit-vector inspired abstract dataflow embedding based on the analogy of graph learning and DFA algorithms.

Transformer models such as CodeBERT [23], LineVul [24], and UniXcoder [26] used a token-based program representation pretrained on a large body of NL-PL pairs, and then fine-tuned for vulnerability detection. Using CFGs, our graph learning only propagates the information along semantically important edges instead of trying to learn the relations of each pair of tokens. Thus, our approach is substantially more efficient. Since we have used a semantic-based embedding, we show that we can improve the performance of token based models. The most recent work, ContraFlow [15], learns embeddings of def-use paths (an output of dataflow analysis), then predicts vulnerability detection using a transformer model. Our work directly emulates dataflow analysis and does not require an expensive pretraining phase.

There were also models that used sequence and CNN architectures. VulDeePecker [38] used BiLSTM on slices considering data dependencies. SySeVR [37] used BiGRU on slices and adds data dependencies. Draper [46] used CNN and Random Forest. However, none of these models integrates dataflow analysis in its algorithm.

Cummins et al. [17] formulated dataflow analyses as supervised learning tasks and applied it for device mapping and algorithm classification; we discuss the differences from our work in-depth in Section 3.3. Other relevant work that explores dataflow analysis and deep learning include: (1) VenkataKeerthy et al. [50] used the output of dataflow analysis, reaching definitions and live variables, to learn flow-aware embeddings; and (2) Bielik et al. [7] and Jeon et al. [32] learned static analysis formulas from a dataset based on a fixed language. None of these works aims to develop a model for vulnerability detection.

8 CONCLUSIONS AND FUTURE WORK

We propose DeepDFA, an efficient graph learning framework and embedding technique for vulnerability detection. Our *abstract dataflow embedding* leverages the idea of *bit-vector* in dataflow analysis and integrates data usage patterns from semantic features: commonly used API calls, operations, constants, and data types that potentially capture the causes of the vulnerabilities. DeepDFA emulates the Kildall method of dataflow analysis using the analogous message-passing algorithm. Our experimental results show that DeepDFA is very efficient. It is trained in 9 minutes and used only 50 vulnerable examples to achieve its top performance. Yet, it still outperformed all non-transformer baselines and generalized the best among all the models. DeepDFA found bugs in real-world programs from DbgBench while neither of the highest-performing baselines, LineVul and UniXcoder, detected any bugs. Importantly, DeepDFA can be used to improve other models. By combining DeepDFA with the top performing models, we surpassed the state-of-the-art performance for vulnerability detection.

In the future, we plan to incorporate other dataflow analyses, e.g., live variable analysis, that have been used for or vulnerability detection [12]. We also plan to explore the application of explanation tools to precisely pinpoint the vulnerability location at specific lines in the code, and evaluate our framework on detecting vulnerabilities in other programming languages.

9 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable feedback. This research is partially supported by the U.S. National Science Foundation (NSF) under Awards #1816352 and #2313054.

REFERENCES

[1] 2021. Browse vulnerabilities by date. https://web.archive.org/web/20211014235218/https://www.cvedetails.com/browse-by-date.php. Accessed November 5 2021.

[2] 2021. IBM Cost of a Data Breach Report 2021. Accessed October 29 2021.

[3] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[5] Miltiadis Allamanis. 2019. The Adverse Effects of Code Duplication in Machine Learning Models of Code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Athens, Greece) *(Onward! 2019)*. Association for Computing Machinery, New York, NY, USA, 143–153. https://doi.org/10.1145/3359591.3359735

[6] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations*. https://openreview.net/forum?id=BJOFETxR-

[7] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2017. Learning a static analyzer from data. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30*. Springer, 233–253.

[8] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning*. Springer.

[9] Marcel Böhme, Ezekiel Olamide Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the Bug and How is it Fixed? An Experiment with Practitioners. In *Proceedings of the 11th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2017)*. 1–11.

[10] Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. 2021. BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection. *Inf. Softw. Technol.* 136, C (aug 2021), 11 pages. https://doi.org/10.1016/j.infsof.2021.106576

[11] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. 2022. MVD: Memory-Related Vulnerability Detection Based on Flow-Sensitive Graph Neural Networks. *arXiv:2203.02660 [cs]* (March 2022). https://doi.org/10.1145/3510003.3510219 arXiv: 2203.02660.

[12] Silvio Cesare. 2013. Bugalyze.com - Detecting Bugs Using Decompilation and Data Flow Analysis. In *BlackHat USA*. 9.

[13] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Transactions on Software Engineering* 48, 9 (2022), 3280–3296. https://doi.org/10.1109/TSE.2021.3087402

[14] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network. *ACM Trans. Softw. Eng. Methodol.* 30, 3, Article 38 (apr 2021), 33 pages. https://doi.org/10.1145/3436877

[15] Xiao Cheng, Guanqin Zhang, Haoyu Wang, and Yulei Sui. 2022. Path-Sensitive Code Embedding via Contrastive Learning for Software Vulnerability Detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 519–531. https://doi.org/10.1145/3533767.3534371

[16] Miles Cranmer, Alvaro Sanchez Gonzalez, Peter Battaglia, Rui Xu, Kyle Cranmer, David Spergel, and Shirley Ho. 2020. Discovering Symbolic Models from Deep Learning with Inductive Biases. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 17429–17442. https://proceedings.neurips.cc/paper_files/paper/2020/filec9f2f917078bd2db12f23c3b413d9cba-Paper.pdf

[17] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F P O'Boyle, and Hugh Leather. 2021. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 2244–2253. https://proceedings.mlr.press/v139/cummins21a.html

[18] Thomas G. Dietterich. 1998. Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms. *Neural Computation* 10, 7 (10 1998), 1895–1923. https://doi.org/10.1162/089976698300017197 arXiv:https://direct.mit.edu/neco/article-pdf/10/7/1895/814002/089976698300017197.pdf

[19] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. HOPPITY: LEARNING GRAPH TRANSFORMATIONS TO DETECT AND FIX BUGS IN PROGRAMS. In *International Conference on Learning Representations*. https://openreview.net/forum?id=SJeqs6EFvB

[20] Y. Ding, S. Suneja, Y. Zheng, J. Laredo, A. Morari, G. Kaiser, and B. Ray. 2022. VELVET: a noVel Ensemble Learning approach to automatically locate VulnErable sTatements. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, Los Alamitos, CA, USA, 959–970. https://doi.org/10.1109/SANER53432.2022.00114

[21] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. 2019. Leopard: Identifying Vulnerable Code for Vulnerability Assessment through Program Metrics. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 60–71. https://doi.org/10.1109/ICSE.2019.00024

[22] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 508–512. https://doi.org/10.1145/3379597.3387501

[23] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[24] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-Based Line-Level Vulnerability Prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 608–620. https://doi.org/10.1145/3524842.3528452

[25] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. arXiv:1704.01212 [cs.LG]

[26] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. arXiv:2203.03850 [cs.CL]

[27] Hazim Hanif and Sergio Maffeis. 2022. VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection. In *2022 International Joint Conference on Neural Networks (IJCNN)*. 1–8. https://doi.org/10.1109/IJCNN55064.2022.9892280

[28] David Hin, Andrey Kan, Huaming Chen, and M. Ali Babar. 2022. LineVD: Statement-Level Vulnerability Detection Using Graph Neural Networks. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 596–607. https://doi.org/10.1145/3524842.3527949

[29] Susan Horwitz. [n. d.]. *Dataflow Analysis*. https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.DATAFLOW.html CS704 Lecture Notes [Accessed 19-09-2023].

[30] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:1704.04861 [cs.CV]

[31] Nathalie Japkowicz. 2000. The class imbalance problem: Significance and strategies. In *Proc. of the Int'l Conf. on artificial intelligence*, Vol. 56. 111–117.

[32] Minseok Jeon, Sehun Jeong, Sungdeok Cha, and Hakjoo Oh. 2019. A Machine-Learning Algorithm with Disjunctive Model for Data-Driven Program Analysis. *ACM Trans. Program. Lang. Syst.* 41, 2, Article 13 (June 2019), 41 pages. https://doi.org/10.1145/3293607

[33] Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Boston, Massachusetts) *(POPL '73)*. Association for Computing Machinery, New York, NY, USA, 194–206. https://doi.org/10.1145/512927.512945

[34] Quoc V. Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. arXiv:1405.4053 [cs.CL]

[35] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2017. Gated Graph Sequence Neural Networks. arXiv:1511.05493 [cs.LG]

[36] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability detection with fine-grained interpretations. (2021), 292–303. https://doi.org/10.1145/3468264.3468597

[37] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2022. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2022), 2244–2258. https://doi.org/10.1109/TDSC.2021.3051525

[38] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. February (2018). https://doi.org/10.14722/ndss.2018.23158

[39] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* abs/2102.04664 (2021).

[40] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. arXiv:1301.3781 [cs.CL]

[41] Sara Moshtari, Ahmet Okutan, and Mehdi Mirakhorli. 2022. A Grounded Theory Based Approach to Characterize Software Attack Surfaces. In *Proceedings of the*

44th International Conference on Software Engineering (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/3510003.3510210

[42] Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. 2022. ReGVD: Revisiting Graph Neural Networks for Vulnerability Detection. In *Deep Learning for Code Workshop*. https://openreview.net/forum?id=BU5eniuWkbq

[43] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. GloVe: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 1532–1543. https://doi.org/10.3115/v1/D14-1162

[44] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506. https://doi.org/10.1145/3394486.3406703

[45] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. Association for Computing Machinery, New York, NY, USA, 49–61. https://doi.org/10.1145/199448.199462

[46] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 757–762. https://doi.org/10.1109/ICMLA.2018.00120

[47] Skipper Seabold and Josef Perktold. 2010. statsmodels: Econometric and statistical modeling with python. In *9th Python in Science Conference*.

[48] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. 2023. An Empirical Study of Deep Learning Models for Vulnerability Detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2237–2248. https://doi.org/10.1109/ICSE48619.2023.00188

[49] Mingxing Tan and Quoc Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 6105–6114. https://proceedings.mlr.press/v97/tan19a.html

[50] S. VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and Y. N. Srikant. 2020. IR2VEC: LLVM IR Based Scalable Program Embeddings. *ACM Trans. Archit. Code Optim.* 17, 4, Article 32 (dec 2020), 27 pages. https://doi.org/10.1145/3418463

[51] Yu Wang, Fengjuan Gao, Linzhang Wang, and Ke Wang. 2020. Learning a Static Bug Finder from Data. *arXiv:1907.05579 [cs]* (March 2020). http://arxiv.org/abs/1907.05579 arXiv: 1907.05579.

[52] Wikipedia. 2021. List of data breaches. https://web.archive.org/web/20211011144237/https://en.wikipedia.org/wiki/List_of_data_breaches. Accessed October 29 2021.

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

[54] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. *Proceedings - IEEE Symposium on Security and Privacy* (2014), 590–604. https://doi.org/10.1109/SP.2014.44

[55] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2A: A Dataset Built for AI-Based Vulnerability Detection Methods Using Differential Analysis. (2021), 111–120. https://doi.org/10.1109/icse-seip52600.2021.00020

[56] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in Neural Information Processing Systems* 32 (2019), 1–11. https://doi.org/10.5555/3454287.3455202