DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection

Yizheng Chen
University of Maryland
yzchen@umd.edu

Zhoujie Ding
UC Berkeley
zhoujie.ding@berkeley.edu

Lamya Alowain
King Abdulaziz City for Science and Technology
lalowain@kacst.edu.sa

Xinyun Chen
Google Deepmind
xinyunchen@google.com

David Wagner
UC Berkeley
daw@cs.berkeley.edu

ABSTRACT

We propose and release a new vulnerable source code dataset. We curate the dataset by crawling security issue websites, extracting vulnerability-fixing commits and source codes from the corresponding projects. Our new dataset contains 18,945 vulnerable functions spanning 150 CWEs and 330,492 non-vulnerable functions extracted from 7,514 commits. Our dataset covers 295 more projects than all previous datasets combined.

Combining our new dataset with previous datasets, we present an analysis of the challenges and promising research directions of using deep learning for detecting software vulnerabilities. We study 11 model architectures belonging to 4 families. Our results show that deep learning is still not ready for vulnerability detection, due to high false positive rate, low F1 score, and difficulty of detecting hard CWEs. In particular, we demonstrate an important generalization challenge for the deployment of deep learning-based models. We show that increasing the volume of training data may not further improve the performance of deep learning models for vulnerability detection, but might be useful to improve the generalization ability to unseen projects.

We also identify hopeful future research directions. We demonstrate that large language models (LLMs) are a promising research direction for ML-based vulnerability detection, outperforming Graph Neural Networks (GNNs) with code-structure features in our experiments. Moreover, developing source code specific pre-training objectives is a promising research direction to improve the vulnerability detection performance.

KEYWORDS

datasets, vulnerability detection, deep learning, large language models

ACM Reference Format:
Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. 2023. DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. In *The 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '23), October 16–18, 2023, Hong Kong, China.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3607199.3607242

1 INTRODUCTION

Detecting software vulnerabilities is crucial to prevent cybercrimes and economic losses, but to date it remains a hard problem. Traditional static and dynamic vulnerability detection techniques suffer from shortcomings. Given the tremendous success of deep learning in image and natural language applications, it is natural to wonder if deep learning can enhance our ability to detect vulnerabilities [4, 15, 18, 25, 33]. However, as we show in this paper, we still need to overcome many challenges before deep learning can achieve great performance for vulnerable source code detection.

For deep learning to be successful, we need a large dataset of vulnerable source code. We release a new open vulnerability dataset for C/C++, DiverseVul. To curate the dataset, we crawl security issue websites, collect vulnerability reports, extract vulnerability-fixing commits for each vulnerability, clone the corresponding projects, and extract vulnerable and nonvulnerable source code from them. Our dataset contains 18,945 vulnerable functions and 330,492 nonvulnerable functions extracted from 7,514 commits, covering 150 CWEs. This is more than twice the size of the C/C++ data from the previous largest and most diverse dataset CVEFixes [2]. Our dataset is more diverse and covers almost 50% more projects than the combination of all previously published datasets. We publicly release the DiverseVul dataset to the community at https://github.com/wagner-group/diversevul.

Our new dataset has enabled us to study the state-of-the-art deep learning methods and gain new insights about promising research directions as well as the challenges for ML-based vulnerability detection. In particular, we study several questions. Does more training data help, or are models saturated? Does the model architecture make a big difference? Is it better to use the state-of-the-art model that relies on code-structure features, or better to use large language models? Is a larger LLM better than a smaller LLM? What are the most promising directions for further improving deep learning for vulnerability detection?

To study the effect of model architectures, we experiment with 11 different deep learning architectures from 4 representative model families: Graph Neural Networks (GNN) [13], RoBERTa [10, 11, 16], GPT-2 [17, 23, 30], and T5 [3, 24, 29]. Much work on deep learning

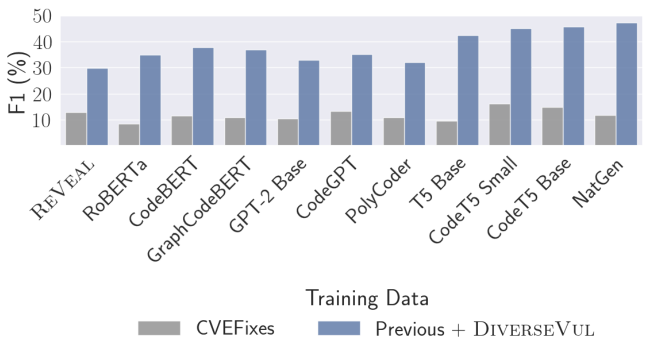

Figure 1: An overview of several of our results. When trained on only the CVEFixes dataset, ReVeal has comparable performance as large language models. If we have enough data (Previous + DiverseVul), large language models (e.g., NatGen) are superior to previous-generation models (e.g., ReVeal, a GNN model with code-structure features), but we need large datasets to see these benefits. LLMs are better able to take advantage of larger datasets than previous-generation models (blue bars vs gray bars). The best LLMs for this task, CodeT5 and NatGen, have been pre-trained with code-specific tasks.

for vulnerability detection used GNNs with code-structure features [4, 18, 33]. We also explore applying large language models (LLMs) to vulnerability detection, as LLMs have achieved state-of-the-art results for natural language processing and code understanding, even though they don't use code-structure features. We study the performance of these models on three datasets: (1) CVEFixes [2], the largest previously published dataset of C/C++ vulnerabilities; (2) the combination of all previously published datasets (Devign [33], ReVeal [4], BigVul [9], CrossVul [19], CVEFixes [2]), deduplicated; (3) the combination of those previous datasets and our DiverseVul (details in Table 3).

Our experiments show that, when evaluating on a prior dataset CVEFixes [2], the model architecture has little effect and LLMs perform about the same as GNNs. In particular, on CVEFixes, the largest previously released dataset, the ReVeal model (a GNN) achieves 12.8 F1 score, vs F1 scores of 8.5–16.3 for LLMs (see Figure 1). One might be tempted to conclude from this that the exact architecture has little effect. However, when evaluating on larger datasets, we can see that this conclusion is reversed: LLMs can perform significantly better than GNNs. In particular, when we combine all previously published datasets together with our DiverseVul, the best LLM achieves F1 score of 47.2, vs 29.8 for ReVeal.

These experiments show that we need large datasets to reliably evaluate deep learning approaches to vulnerability detection, as the relative performance of different architectures shifts radically as we increase the amount of training data available: a 5× increase in the amount of training data (from CVEFixes to all datasets) improved the performance of our best model from 10.5 to 48.9 F1 score. They suggest that LLMs are better able to make use of large datasets than GNNs: larger datasets improve the performance of ReVeal only modestly, but improve the performance of LLMs significantly. However, our experiments suggest that the performance gain from gathering more data may have stagnated. By adding our dataset to the combination of previous datasets, we can improve the test performance on 7 models out of 11. However, for the 3 best-performing models, either we don't see improvement or the improvement is small (details in Section 4.2).

Unfortunately, the state-of-the-art deep learning techniques are still not ready for vulnerability detection yet. Our best model has 47.2% F1 score, 43.3% true positive rate, and 3.5% false positive rate. The false positive rate is still far too high for the model to be practically useful. A project might contain tens of thousands of functions, and this false positive rate corresponds to hundreds of false positives, which is more than most analysts are likely to be willing to wade through [1].

Despite the challenges, Figure 1 suggests that large language models (LLMs) may be superior for deep learning based vulnerability detection. In previous papers, researchers believe that GNN with code-structure features is promising for vulnerability detection [4, 18, 33], since it combines domain knowledge with deep learning. In contrast, our results show that large language models (RoBERTa, GPT-2, and T5 families) significantly outperform the state-of-the-art GNN, especially when training with more data. In particular, CodeT5 models (CodeT5 Small, CodeT5 Base, NatGen) are the best.

Contrary to the common belief that model size is the most important factor for LLMs to perform well, our results show that the most important factor may be how the LLM is trained. Pretraining on code understanding tasks appears to offer large improvements. For example, CodeT5 Small is pretrained to predict variable and function names, and it can achieve an average of 8 percentage points higher F1 score than models that are twice its size but were not pretrained on code. Surprisingly, we found that pretraining tasks that are effective for natural language do not help vulnerability detection much. Instead, it appears we need code-specific pretraining tasks. We think that developing better code-specific pretraining tasks is a promising research direction for improving deep learning based vulnerability detection.

Moreover, we identify an important generalization challenge for the deployment of deep learning based models. To deploy a model we need to detect vulnerabilities from new software projects that do not appear in the training set. We found that deep learning models perform very poorly in this setting. In particular, past work has split data into training and test sets by a random split of the vulnerabilities, without regard to which project each vulnerability appears in. However, in practice, we often want to run a vulnerability detection tool on a new project, so there won't be any vulnerabilities from that project in the training set. To evaluate the performance of deep learning in this setting, we set aside a held-out set of projects, which we call "unseen projects"; we train on vulnerabilities from the other projects ("seen projects"), and then test on vulnerabilities from unseen projects. The performance of all models on unseen projects decreases significantly, e.g., from a F1 score of 49% on seen projects to only 9.4% on unseen projects. The cause is unclear; perhaps the model is overfitting to patterns or coding idioms that are specific to the particular projects that appear in the training set. This generalization failure is likely to be a significant barrier to deploying deep learning vulnerability detection in practice. We hope future research will explore how to address this problem. We suggest a simple intervention to use class weights in the training loss, that takes a small step in this direction, but the gap remains very large and more work is needed.

Lastly, we quantify the label noise in our dataset as well as previous datasets. Label noise is a significant challenge for ML-based vulnerability detection research. To extract vulnerable functions from vulnerability-fixing commits, following the state-of-the-art approach (used by Devign [33], ReVeal [4], BigVul [9], CrossVul [19], CVEFixes [2]), we label functions that were changed by these commits as vulnerable. To understand the label accuracy of such labeling approach, we randomly sample 50 vulnerable functions from our dataset, and another 50 vulnerable functions from the union of three datasets that collect commits from NVD (BigVul, CrossVul, and CVEFixes). Then, we manually analyze the vulnerability and the labeled vulnerable functions. Our results find that the vulnerable function label in DiverseVul is 60% accurate, which is 24% more accurate than the union of CVEFixes, BigVul and CrossVul but still containing many label errors. The main challenges are vulnerabilities that are spread across multiple functions and changes to non-vulnerable functions in vulnerability-fixing commits. We hope our work takes the first step towards understanding the label noise issue and highlights the need for deeper investigation of the impact of label noise.

We make the following contributions in this paper:

- We release DiverseVul, a new C/C++ vulnerable source code dataset. Our dataset is 60% larger than the previous largest dataset for C/C++, and the most diverse compared to all previous datasets.
- We study 11 model architectures from 4 different model families. Our results show that large language models outperform the state-of-the-art graph neural network for deep learning based vulnerability detection, and developing source-code specific pretraining objectives is a promising research direction.
- We identify challenges of deep learning for vulnerability detection. In particular, we highlight the difficulty of generalizing to unseen projects outside the training set.
- We assess label noise in our dataset and previous datasets that rely on vulnerability-fixing commits.

2 RELATED WORK

In this section, we analyze previous public vulnerable source code datasets for C/C++, their labeling methods, and how they are used by related works on deep learning for vulnerability detection.

Synthetic Datasets: SATE IV Juliet [22] and SARD [21] are common synthetic datasets used by previous papers [15, 18, 25]. SARD expands on the Juliet v1.0 test suite and contains test cases for multiple programming languages. The test cases are highly accurate, and contain a variety of CWEs. However, they are constructed in isolation using known vulnerable patterns, which are designed to evaluate static and dynamic analysis tools. They don't fully capture the complexities of vulnerabilities within the real-world projects. The VulDeePecker [15] dataset focuses on only two CWEs. They selected vulnerabilities from 19 projects according to CVE information from the National Vulnerability Database (NVD) [20], and also combined SARD [21] test cases from these two CWEs. Both VulDeePecker and SARD are semi-synthetic datasets.

Static Analyzer Labels: The Draper [25] dataset generated labels using alerts from three static analyzers: Clang, Cppcheck, and Flawfinder. Some categories of alerts were labeled as vulnerable, and some are mapped to non-vulnerable. The labeled dataset is at the function granularity. The quality of the label is unknown, but the label accuracy of static analyzers tend to be low. D2A [32] used differential analysis on the static analyzer (Infer) output over six open-source repositories. Given thousands of version pairs for a github repository, if the static analyzer generates an alert for the version before a git commit, but not after the commit, then D2A treats the commit as fixing a vulnerability. For the remaining alerts, D2A labels them as unrelated to vulnerabilities.

Manual Labeling: The Devign [33] dataset was labeled by three security researchers. They first used keywords to find commits that likely fixed vulnerabilities and commits unrelated to vulnerabilities from four repositories. Then, for the first category, three security researchers manually reviewed these commits by majority vote to determine which fix security vulnerabilities. Given labels for each commit, Devign extracts the changed function before the commit as the data sample, and labels it as vulnerable or non-vulnerable according to the label of the commit. The authors of Devign released data for two repositories, `FFMPeg` and `Qemu`. This dataset has high quality labels, but manual labeling was very expensive, costing around 600 man-hours.

Security Issues: Several prior datasets were generated by crawling security issues to identify vulnerability-fixing commits. The ReVeal [4] dataset was labeled using the patches to known security issues at Chromium security issues and Debian security tracker. ReVeal considers the changed functions before a security patch (commit) as vulnerable, after the patch as non-vulnerable, and all unchanged functions as non-vulnerable. In comparison, our dataset DiverseVul has 18K vulnerable functions, which is 11× the size of ReVeal (Table 3).

BigVul [9], CrossVul [19] and CVEfixes [2] collect vulnerability-fixing commits from Common Vulnerabilities and Exposures (CVE) records in the NVD [20]. In particular, CVEFixes covers all published CVEs up to 27 August 2022. CVEfixes and CrossVul datasets cover multiple programming languages, and we use their C/C++ data in this paper. These three datasets cover a wide range of projects and CWEs. In comparions, our dataset contains more projects, more CWEs, and double the number of vulnerability-fixing commits.

A few other vulnerable source code datasets in C/C++ do not provide vulnerable functions, and therefore we did not include them in our experiments. For example, AOSP [5] collected commits fixing CVEs from the security bulletin of Android Open Source Project (AOSP), which contain patches to vulnerabilities in Android OS, the Linux kernel, and system on chip manufacturers. PatchDB [28] provides patch information, i.e., code diffs, but does not provide enough information to identify the project or git repository it came from and thus does not let us reconstruct the full code of the changed funcction.

Security issues are effective at identifying vulnerability-fixing commits, as they are based on manual analysis from developers.

They are also representative of in-the-wild vulnerabilities in real-world projects. Therefore, we also collect our new dataset DiverseVul by crawling security issues. Compared to all previous datasets, DiverseVul is the most diverse one, covering the most number of projects. In particular, DiverseVul has vulnerabilities from 295 new projects that have not been collected by any of the previous real-world datasets (Table 3).

DL for Vulnerable Source Code Detection: Previous papers have used LSTM [15], CNNs and RNNs [25], Bidirectional RNNS [14], and Graph Neural Networks [4, 18, 33] to detect vulnerable source code. A recent paper from Thapa et al. [27] shows that on the VulDeePecker [15] dataset spanning two CWEs, large language models outperform BiLSTM and BiGRU models. However, they did not compare against Graph Neural Networks (GNN). GNNs represent programs as graphs that contain useful domain knowledge for vulnerability detection. ReVeal [4] used features obtained from the code property graph [31], and VulChecker [18] proposed a new enriched program dependence graph. These papers used relatively small datasets such as ReVeal and Juliet. If we train the models with larger datasets, it is not clear whether GNN with code-structure features is still effective compared to large language models.

3 DATA COLLECTION

Our goal is to collect high-quality vulnerability-fixing commits from a diverse set of real-world projects. We focus on collecting data from security issues, since they reflect high-quality labels from a community of developers and security analysts. We start by identifying 29 security issue websites, and then narrow it down to 2 websites with most git system commits [1]. From these websites, we crawl the issue title, body, and relevant git commit URLs. Since developer's discussions may reference both vulnerability-fixing commits and vulnerability-introducing commits, we use two heuristics to exclude vulnerability-introducing commits. First, we exclude all commit URLs mentioned in comments containing keywords "introduced" and "first included"; and second, we manually go over all commits that changed at least 10 functions and exclude ones that introduced vulnerability. We keep the remaining commits in our dataset.

Next, we parse the git commit URLs to extract the projects and commit IDs. We clone the projects and extract the commits from these projects. We identify the C/C++ related code files in the commits. Then, we extract all functions that were changed in these commits, and also functions that did not change in the files. Same as ReVeal [4], we label the before-commit version of a changed function to be vulnerable, and the after-commit version to be non-vulnerable. We label all unchanged functions in the related code files to be non-vulnerable. Like prior work, we deduplicate functions by their MD5 hashes, and we do not normalize the code before deduplication. We keep track of the set of unique MD5s when processing the functions. We process all the vulnerable functions before the nonvulnerable ones. If the MD5 of a function already exists in this set, we do not include the function again in the data. In total, we have collected 7,514 commits from 797 projects, which result in 18,945 vulnerable functions and 330,492 non-vulnerable functions, covering 150 CWEs. Table 1 shows the top 10 projects and the top 10 CWEs in DiverseVul with the most number of vulnerability-fixing commits. Note that CWE-703 "Improper Check or Handling of Exceptional Conditions" is not on the list of MITRE top-25 CWEs.

[1] `snyk.io` and `bugzilla.redhat.com`.

Project	# Commits
linux	1,458
ImageMagick	330
php-src	301
openssl	261
tensorflow	243
qemu	205
linux-2.6	179
vim	134
FFmpeg	134
tcpdump	112

(a)

CWE	# Commits
CWE-787	2,896
CWE-125	1,869
CWE-119	1,633
CWE-20	1,315
CWE-703	1,228
CWE-416	1,005
CWE-476	975
CWE-190	783
CWE-200	747
CWE-399	509

(b)

Table 1: Top 10 projects and CWEs in DiverseVul and the corresponding number of vulnerability-fixing commits.

For issue titles that mention the CVE number, we query the National Vulnerability Database API to obtain the CWE information for the issue and the corresponding commit. For issues with developer annotated vulnerability category, we manually map them to top 25 most popular CWEs. About 85% of our data can be mapped to 150 CWE categories. We do not specifically address hierarchical CWEs. Depending on the query result from the NVD Database, a CVE number could be mapped to multiple CWEs.

4 EXPERIMENTS

In this section, we study how our new dataset can improve the performance of deep learning based vulnerability detection. We study 11 model architectures from 4 model families. We also discuss insights learned from these experiments.

4.1 Model Architectures

We study 4 model families, where 3 families are transformer-based large language models (LLM). Within each LLM family, there are different variants of the model pretrained using different objectives. Table 2 summarizes the number of parameters for all model architectures.

4.1.1 Graph Neural Network. Within the Graph Neural Network (GNN) family, we choose to reproduce a representative previous work ReVeal [4].

Given a function, the ReVeal model constructs a graph to represent the function, computes the embedding vector of the graph, and classifies the vector as vulnerable or nonvulnerable. Specifically, the graph representation for the function is a code property graph [31] (CPG). CPG combines Abstract Syntax Tree (AST), Control Flow Graph (CFG), Data Flow Graph (DFG), and Program Dependence Graph (PDG). Each node has the corresponding source code and type, and each edge has a type. The embedding of the graph is a sum of embeddings of the nodes in the graph. To learn the node embeddings, ReVeal uses Gated Graph Neural Networks (GGNN) [13]

Model Family	Model Architecture	# Parameters
GNN	ReVeal	1.28M
RoBERTa	RoBERTa	125M
	CodeBERT	125M
	GraphCodeBERT	125M
GPT-2	GPT-2 Base	117M
	CodeGPT	124M
	PolyCoder	160M
T5	T5 Base	220M
	CodeT5 Small	60M
	CodeT5 Base	220M
	NatGen	220M

Table 2: The number of parameters for different models.

to recursively update the embeddings of the nodes. The initial embedding of a node is a concatenation of Word2Vec embedding of the code and the categorical type vector. Then, the GGNN training procedure uses the message passing mechanism to update each node embedding according to the node's neighbors in the graph. Finally, after training the GGNN, ReVeal adds two fully-connected layers, rebalances the training set, to learn the final classifier. The total number of parameters of the ReVeal model is 1.28M.

4.1.2 RoBERTa Family. We select three model achitectures from the RoBERTa family: RoBERTa [16], CodeBERT [10], and GraphCodeBERT [11]. All of them have 12 layers of Transformer encoders, 768 dimenional hidden states, 12 attention heads, and 125M model parameters in total. The common pretraining objective for this family is masked language modeling (MLM). The MLM pretraining process randomly masks a percentage of tokens within the input tokens, effectively removing them, and the training goal is to predict the missing tokens.

RoBERTa [16] is an extension of BERT [8] that makes changes to important hyperparameters, including removing the pretraining objective of predicting the next sentence, as well as using larger mini-batches and learning rates during training. RoBERTa was pretrained on a union of five datasets: BookCorpus, English Wikipedia, CC-News, OpenWebText, and Stories.

CodeBERT [10] pretrains the model using the CodeSearchNet [12] dataset containing 2.3M functions from six programming languages (Go, Java, JavaScript, PHP, Python, and Ruby). CodeBERT performs MLM pretraining and replaced token detection pretraining. During pretraining, each input is a pair of natural language description and source code, where the text describes the meaning of the code. The MLM pretraining in CodeBERT makes sure that tokens from both the natural language part and the source code part are masked out, and the replaced token detection corrupts both parts of the input as well. CodeBERT outperforms RoBERTa on two downstream tasks, natural language code search and code documentation generation.

GraphCodeBERT [11] also uses the CodeSearchNet [12] training datasets. In addition to having the natural language description and the source code parts of the input, GraphCodeBERT pretraining also constructs a third part of the input that captures the data flow between variables in the source code. In addition to MLM pretraining, GraphCodeBERT proposes two new pretraining objectives: edge prediction and node alignment. The edge prediction task maximizes the dot product between embeddings of two nodes if there is an edge, and the node alignment task maximize the dot product between embeddings of the code token and variable token if the variable represents the code token. Over benchmark datasets, GraphCodeBERT outperforms CodeBERT and RoBERTa on code clone detection, code translation, and code refinement tasks.

Note that the training dataset of CodeBERT and GraphCodeBERT does not have programs written in C/C++.

4.1.3 GPT-2 Family. We select three model architecures from the GPT-2 family: GPT-2 Base [23], CodeGPT [17], and PolyCoder [30]. They have 12 layers of Transformer decoders, 768 dimentional hidden embeddings, and 12 attention heads. The size of the models are in Table 2, ranging from 117M to 160M. The common pretraining objective for this family is causal language modeling, i.e., next token prediction. How well a model is pretrained on the causal language modeling is measured by perplexity. A lower perplexity value indicates a better model.

GPT-2 [23] was pretrained on an unreleased WebText dataset, which was collected by scraping web page links on Reddit.

CodeGPT [17] uses the same training objective and architecture of GPT-2, but different training data. The authors select Python and Java codes from CodeSearchNet [12] as the training set, and release several variants of the pretrained CodeGPT models. In this paper, we use an adapted version of CodeGPT pretrained on Java codes. The CodeGPT model was initialized from GPT-2 weights, and then pretrained using Java codes from CodeSearchNet using the next token prediction task. Note that there is no C/C++ programs in the training set.

PolyCoder [30] uses the same model architecture and pretrianing objective as GPT-2, but pretrains the model from scratch. The authors pretrained the model with data from GitHub containing both source code and natural language comments within the code files. They cloned a total of 147,095 projects, that are the most popular repositories of 12 popular programming languages with at least 50 stars. Their training data contains over 24K repositories in C/C++. The authors curate an evaluation datasets of codes from unseen repositories. On C programming language, PolyCoder achieves the lowest perplexity value, compared to GPT-Neo, GPT-J, and Codex.

4.1.4 T5 Family. We select four model achitectures from the T5 family: T5 Base [24], CodeT5 Base, CodeT5 Small [29], and NatGen [3]. All models have encoder-decoder Transformer layers. CodeT5 Small has 6 encoder layers and 6 decoder layers, 512 dimensional hidden states, 8 attention heads, and 60M parameters. The other models have 12 encoder layers and 12 decoder layers, 768 dimensional hidden states, 12 attention heads, and 220M parameters.

T5 [24] pretrains the model using the masked language modeling objective. In particular, T5 pretraining procedure randomly masks spans of tokens. The pretraining dataset is C4 (Colossal Clean Crawled Corpus). The authors curate the C4 dataset by processing the Common Crawl dataset to get hundreds of gigabytes of clean English text.

CodeT5 [29] uses the same underlying transformer architecture as T5. We consider two model sizes in our experiments: CodeT5 Base and CodeT5 small. The CodeT5 Small is the smallest LLM, with one third the model size of other T5 based models, and roughly half the model size of RoBERTa and GPT-2 family models. CodeT5 was pretrained on on both CodeSearchNet data and additional C/C# projects from GitHub. In addition to the masked span prediction objective, CodeT5 utilizes the knowledge about whether a token is an identifer (a variabel name or a function name) and designs two new pretraining tasks. The new pretraining tasks are, masked identifier prediction (masking all identifiers) and identifier tagging (predict whether a token is an identifier).

NatGen [3] proposes a new pretraining objective called "naturalizing" pretraining. The naturalizing pretraining is similar to a code editing process, that takes some weird synthetic code and tranform that into developer-readable code. The authors generate un-natural code by semantic preserving code transformations including adding dead code, changing a while loop to a for loop without variable initialization, renaming variables, and inserting confusing code element, etc. Then, the pretraining objective asks the model to naturlize the code to the original developer-friendly form. The NatGen model starts the pretraining from the CodeT5 Base weights, and then continues the pretraining process using their new pretraining objective. Doing well on the naturalizing pretraining objective requires the model to understand the code well. Compared to CodeT5, NatGen improves the performance over various downstream tasks such as code translation, text to code generation, and bug repair.

4.2 Model Performance with More Data

4.2.1 Dataset Setup. Deep learning models perform well when they are trained on a lot of data. Therefore, we combine non-synthetic datasets with high-quality vulnerability labels from real-world projects, including Devign, ReVeal, BigVul, CrossVul, and CVEFixes. We then combine them with DiverseVul and remove duplicate samples to create the Previous + DiverseVul dataset, as shown in Table 3.

Table 3 presents the statistics for each of the previous five datasets, as well as our dataset, DiverseVul, and the merged datasets. Compared to all previous datasets, DiverseVul includes a larger number of projects, more CWEs, more vulnerable functions, and more vulnerability-fixing commits. Specifically, DiverseVul contains 18,945 vulnerable functions, of which 16,109 have CWE information, more than twice the number in any previous dataset. Having more data associated with CWE information will provide us with a more comprehensive understanding of model prediction results. The last two rows in Table 3 show the unique new data provided by DiverseVul in the merged datasets after deduplicating samples. Comparing Previous and Previous + DiverseVul datasets, we can see that DiverseVul contains 295 new projects that do not exist in any of the previous datasets. Moreover, DiverseVul provides 10,845 unique new vulnerable functions.

For our experiments, we randomly select 80% of the samples from the Previous + DiverseVul dataset as the training set, 10% as the validation set, and 10% as the test set. We also construct the Previous training and validation sets that only contain the previous five datasets, and training and validation sets that only contain CVEFixes data. This allows us to train models with different amounts of data and evaluate how much adding more data helps in improving the model's performance to predict the same test set from Previous + DiverseVul.

4.2.2 Results. For each model architecture in Table 2, we train three models, using CVEFixes, Previous, and Previous + DiverseVul training datasets. We train the ReVeal models from scratch, and we fine tune the large language models (LLMs) for the vulnerability detection task from pretrained model weights. This gives us 33 models in total. The detailed training setups in our experiments can be found in Appendix A.

Table 4 shows the performance of the models over the same test set from Previous + DiverseVul. The following summarizes the results.

Result 1: When trained on all available data, large language models significantly outperform the state-of-the-art GNN-based ReVeal model. When trained on all available data (Previous + DiverseVul), LLMs perform significantly better than the ReVeal model: the ReVeal model achieves a 29.76 F1 score, while LLMs achieve F1 scores from 31.96 to 47.15. The best LLM performs significantly better than ReVeal on this large training set. Comparing between ReVeal and LLMs is arguably unfair since ReVeal has 1–2 orders of magnitude fewer parameters than LLMs. We do not know whether a larger GNN could be competitive with LLMs. Unfortunately, even the best-performing model, NatGen, is not yet suitable for deployment in vulnerability detection, with a 3.47% false positive rate and a 47.15% F1 score. This false positive rate is still too high to be practical, and the F1 score is still low. Nevertheless, we believe that large language models hold promise for deep learning-based vulnerability detection.

Interestingly, LLMs require a large amount of training data to surpass ReVeal. When trained solely on CVEFixes data, a much smaller training set, there is no clear advantage of LLMs over GNN-based ReVeal model, and ReVeal is even better than 6 LLMs (out of 10) in this setting.

Result 2: Within the three base LLM models, T5 Base performs better than RoBERTa and GPT-2 Base for vulnerability detection. RoBERTa only uses encoders, GPT-2 only uses decoders, and T5 uses encoder-decoder Transformer layers. When trained on Previous + DiverseVul, T5 Base has a test F1 score that is 7.35% and 9.3% higher than RoBERTa and GPT-2 Base, respectively. Thus, an encoder-decoder architecture might have an advantage over a decoder/encoder only architecture.

Result 3: Pretraining on code does not lead to significant improvements in vulnerability prediction, if we only use natural language pretraining tasks. The code models CodeBERT, GraphCodeBERT, CodeGPT, PolyCoder are not significantly better than the corresponding text models RoBERTa and GPT-2 Base. Specifically, when trained on the Previous dataset, CodeBERT and GraphCodeBERT perform similarly to RoBERTa. When trained on the Previous + DiverseVul dataset, CodeBERT and GraphCodeBERT improve the F1 score by up to 2.8% compared to RoBERTa. On the other hand, when trained on Previous dataset, CodeGPT and

Dataset	# Projects	# CWEs	# Functions	# Vul Func	# Vul Func with CWE Info	# Commits
Devign	2$^{\triangledown}$	N/A	26,037	11,888	N/A	N/A
ReVeal	2$^{\diamond}$	N/A	18,169	1,664	N/A	N/A
BigVul	348	91	264,919	11,823	8,783	3,754
CrossVul*	498	107	134,126	6,884	6,833	3,009
CVEFixes*	564	127	168,089	8,932	8,343	3,614
DiverseVul	**797**	**150**	**330,492**	**18,945**	**16,109**	**7,514**
Previous†	638	140	343,400	30,532	14,159	17,956
Previous + DiverseVul	**933**	**155**	**523,956**	**41,377**	**22,382**	**21,949**

†: We aggregate previous five datasets by combining and deduplicating samples from Devign, ReVeal, BigVul, CrossVul, and CVEfixes.
*: CVEfixes and CrossVul are multi-language datasets. We report numbers for C/C++ code.
$^{\triangledown}$: Devign authors released data from two repositories: `FFMPeg+Qemu`. $^{\diamond}$: Chromium and Debian packages.

Table 3: Statistics about previous five datasets, DiverseVul, merged Previous dataset, and Previous + DiverseVul.

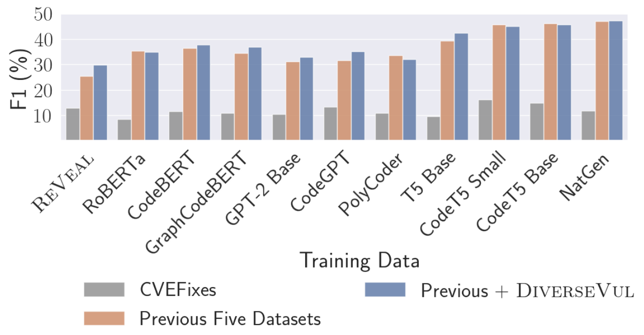

Figure 2: We visualize the performance of models that are trained on CVEFixes, Previous, and Previous + DiverseVul. Adding DiverseVul to the merged Previous dataset helps improve the test performance for 7 models out of 11. It does not help the CodeT5 models.

PolyCoder have up to 2.3% higher F1 scores than GPT-2; but when trained on Previous + DiverseVul, PolyCoder performs worse than GPT-2. Our findings suggest that pretraining models on code using MLM or next token prediction techniques does not yield significant improvements in detecting C/C++ vulnerabilities. While CodeBERT, GraphCodeBERT, and CodeGPT have not pretrained on C/C++, PolyCoder has pretrained over C/C++ code for next token prediction, which still does not help detecting C/C++ vulnerabilities.

Result 4: Code-specific pretraining tasks on C/C++ make a big difference in improving vulnerability detection performance. The two CodeT5 models and the NatGen model have the best F1 scores. They are pretrained using code-specific pretraining tasks on C/C++. CodeT5 models use identifier-aware pretraining tasks: masked identifier prediction and identifier tagging. NatGen does additional code naturalizing pretraining on top of CodeT5, such as removing dead code and renaming variables. These pretraining tasks ask the model learn about basic code understanding, which significantly improves the fine-tuned model performance for vulnerability detection task. Note that GraphCodeBERT also does some code-specific pretraining to learn embeddings from a pair of variables with data flow to have large dot product value. However, since it did not train on C/C++ data, it is unknown whether such pretraining task is effective for vulnerability prediction.

Result 5: Code-specific pretraining task is more important than the model size. Among the best three models in Table 4 (CodeT5 Small, CodeT5 Base, NatGen), the CodeT5 Small model has only 60M parameters, half of the size of RoBERTa models and GPT-2 models, and less than one third the size of other T5 models. However, CodeT5 Small performs very similar to the largest CodeT5 Base and NatGen models, and it performs better than all the other models. Contrary to the belief that larger models tend to produce better performance, our results show that code-specific pretraining task is more important than the model size for vulnerability detection.

Result 6: Performance gain from collecting more datasets may have saturated. Figure 2 visualizes how much training on DiverseVul + Previous data helps improve the vulnerability detection performance, compared to Previous data. Adding DiverseVul to the training set improves the F1 score for 7 models by 2.4% on average, compared to only training with the Previous dataset. However, it does not help the best performing CodeT5 models, and it only helps NatGen modestly. Even though we see a big improvement to model performance by training on the merged Previous datasets compared to only training on CVEFixes, collecting a different dataset may not further improve that.

4.3 Dataset Volume

4.3.1 Dataset Setup. We want to measure the effect of data volume on model performance for vulnerability detection. We run the following experiment ten times. For each run, we randomly split the Previous + DiverseVul into training, validation, and test sets. Then, we simulate the effect of different data volume by subsampling the training and validation sets. Specifically, we randomly sample 10% to 90% of the training and validation data from the full training and validation data of Previous + DiverseVul. Then, we train the

Model Family	Model Arch	Pretrain on Code	Pretrain on C/C++	Code-Specific Pretrain Task	Training Set	Test on Prev + DiverseVul (%) Acc	Prec	Recall	F1	FPR
GNN	ReVeal				CVEFixes	82.12	11.56	14.37	12.81	11.06
					Previous	86.30	25.35	25.63	25.49	7.59
					Prev + DiverseVul	82.81	23.75	39.83	29.76	12.87
RoBERTa	RoBERTa				CVEFixes	91.71	34.24	4.85	8.50	0.80
					Previous	90.98	40.97	31.11	35.37	3.86
					Prev + DiverseVul	91.68	46.02	28.22	34.98	2.85
	CodeBERT	✔			CVEFixes	91.62	35.64	6.98	11.67	1.09
					Previous	91.07	41.83	32.20	36.39	3.86
					Prev + DiverseVul	90.48	39.25	36.54	37.85	4.87
	GraphCodeBERT	✔		✔	CVEFixes	91.76	38.28	6.35	10.89	0.88
					Previous	91.65	45.71	27.61	34.43	2.83
					Prev + DiverseVul	90.32	38.18	35.51	36.79	4.96
GPT-2	GPT-2 Base				CVEFixes	91.45	31.02	6.37	10.57	1.22
					Previous	91.80	46.62	23.46	31.21	2.32
					Prev + DiverseVul	91.73	46.18	25.71	33.03	2.58
	CodeGPT	✔			CVEFixes	90.77	26.22	8.98	13.38	2.18
					Previous	91.59	44.51	24.48	31.58	2.63
					Prev + DiverseVul	91.36	43.48	29.62	35.23	3.32
	PolyCoder	✔	✔		CVEFixes	91.12	26.56	6.78	10.81	1.62
					Previous	91.28	42.44	27.66	33.49	3.23
					Prev + DiverseVul	91.97	48.76	23.78	31.96	2.15
T5	T5 Base				CVEFixes	91.57	32.23	5.65	9.61	1.02
					Previous	92.15	50.80	32.15	39.38	2.68
					Prev + DiverseVul	91.96	49.14	37.17	42.33	3.32
	CodeT5 Small	✔	✔	✔	CVEFixes	90.89	30.03	11.18	16.29	2.24
					Previous	91.98	49.34	42.53	45.68	3.76
					Prev + DiverseVul	91.85	48.41	42.22	45.10	3.88
	CodeT5 Base	✔	✔	✔	CVEFixes	91.41	34.76	9.39	14.79	1.52
					Previous	92.16	50.68	42.46	46.20	3.56
					Prev + DiverseVul	92.11	50.36	41.81	45.69	3.55
	NatGen	✔	✔	✔	CVEFixes	91.64	36.17	7.07	11.83	1.08
					Previous	92.30	51.81	42.92	46.94	3.44
					Prev + DiverseVul	**92.30**	**51.81**	**43.25**	**47.15**	**3.47**

Table 4: We evaluate the models on the same test set from Previous + DiverseVul. There isn't a big difference between model performance across different architectures if we only train on the CVEFixes dataset. However, if we train on larger datasets, large language models significantly outperform the GNN-based ReVeal model. Among them, CodeT5 Small, CodeT5 Base, and NatGen models have the highest F1 scores. We highlight the row with the highest F1 score in bold. Pretraining the model using code-specific pretraining task over C/C++ is very effective.

models, and evaluate them on the same original test set without subsampling.

4.3.2 *Results.*

We fine tune 100 CodeT5 Small models on different dataset setups from 10 experiment runs. Within each run, we evaluate the models on the same final test set from the Previous + DiverseVul, and train 10 models by using different percentages of training and validation data. Figure 3 plots the average and 95% confidence interval for the test F1 score, when a model is fine tuned from a corresponding dataset setup.

Result 7: Increasing the volume of the training dataset from the same distribution helps vulnerability detection. Our results show that training on a larger dataset *from the same distribution* can improve the test performance. Figure 3 shows an upward trend of better test F1 score as the volume of training data increases. If we know the test data distribution ahead of the model

Model Family	Model Arch	Pretrain on Code	Pretrain on C/C++	Code-specific Pretrain task	Training Set	Test on Unseen Projects (%) Acc	Prec	Recall	F1	FPR
GNN	ReVeal				Previous	82.88	5.06	20.92	8.15	14.78
					Prev + DiverseVul	85.88	5.67	18.46	8.67	11.58
RoBERTa	RoBERTa				Previous	94.69	6.20	3.23	4.25	1.85
					Prev + DiverseVul	95.59	10.46	2.78	4.40	0.90
	CodeBERT	✔			Previous	94.94	9.53	4.57	6.17	1.64
					Prev + DiverseVul	**94.19**	**13.34**	**10.80**	**11.94**	**2.65**
	GraphCodeBERT	✔		✔	Previous	95.32	4.64	1.45	2.21	1.12
					Prev + DiverseVul	94.74	12.48	7.35	9.25	1.95
GPT-2	GPT-2 Base				Previous	94.92	6.19	2.78	3.84	1.60
					Prev + DiverseVul	95.06	9.82	4.34	6.02	1.51
	CodeGPT	✔			Previous	94.32	5.98	3.79	4.64	2.25
					Prev + DiverseVul	94.47	9.86	6.35	7.72	2.19
	PolyCoder	✔	✔		Previous	95.41	8.54	2.67	4.07	1.08
					Prev + DiverseVul	92.73	10.25	12.81	11.39	4.24
T5	T5 Base				Previous	95.67	20.21	6.35	9.66	0.95
					Prev + DiverseVul	96.16	34.00	5.68	9.73	0.42
	CodeT5 Small	✔	✔	✔	Previous	95.02	12.21	5.90	7.96	1.60
					Prev + DiverseVul	94.91	13.35	7.24	9.39	1.78
	CodeT5 Base	✔	✔	✔	Previous	96.21	32.32	3.56	6.42	0.28
					Prev + DiverseVul	95.56	18.03	6.12	9.14	1.05
	NatGen	✔	✔	✔	Previous	95.48	17.86	6.68	9.72	1.16
					Prev + DiverseVul	95.49	17.38	6.35	9.30	1.14

Table 5: We randomly choose 95 projects as unseen projects for testing. The remaining projects are used for training. We train each model on seen projects and test them on unseen projects. We highlight the row with the highest F1 score in bold. Overall, the F1 scores show that these models have poor generalization performance on unseen projects. Adding DiverseVul to Previous training set helps improve the generalization performance for all models except NatGen.

deployment time, collecting more training data from that distribution might further improve the performance on vulnerability detection.

4.4 Generalization

4.4.1 Dataset Setup. In a real-world deployment scenario, a vulnerability detection model needs to predict vulnerable source code in new developer projects that it has not been trained on. Therefore, we would like to test a model's performance on unseen projects.

We randomly select 95 unique projects from the merged Previous dataset as the unseen projects test set, to evaluate all models in this experiment. Then, the remaining projects are treated as seen projects in both training set and validation set. For both Previous and Previous + DiverseVul datsets, we randomly sample 90% of the seen projects as the training set, and 10% remaining projects are the validation set. The training and validation sets of Previous + DiverseVul are supersets of these from Previous.

4.4.2 Results. We train ReVeal and fine tune each LLM on the seen projects training set from Previous and Previous + DiverseVul, resulting in 22 models in total. We make sure that these models have been trained well, since they have achieved validation performance similar to training performance. Table 5 shows the test performance of these models over unseen projects.

The F1 scores of all models on unseen projects are very low. The best models are CodeBERT, PolyCoder, CodeT5 Small, CodeT5 Base models trained on Previous + DiverseVul, and NatGen model trained on Previous seen projects. Adding DiverseVul to Previous training set helps improve the generalization performance for all models except NatGen. One recent, concurrent work [26] also observed a significant performance drop when testing on unseen projects. In our experiment, we have included hundreds of more projects in the training set than [26], but we still observe the poor generalization results.

Result 8: There is a significant challenge for deep learning models to generalize to unknown test projects on the vulnerability detection task. A popular use case of AI for Code is the GitHub CoPilot, where the AI model suggests ways to complete code to developers when they are writing code. If AI for deep learning detection is also a coding assistant, it needs to suggest potential

Model Arch	Scheme	Train on Seen Projects Test on Unseen Projects (%)					Train on Random Samples Test on Random Samples (%)				
		Acc	Prec	Recall	F1	FPR	Acc	Prec	Recall	F1	FPR
CodeBERT	No weight	94.19	13.34	10.8	11.94	2.65	90.48	39.25	36.54	37.85	4.87
	Project Balanced	95.09	11.6	5.23	7.21	1.51	90.7	34.43	18.97	24.46	3.11
	Weighted Soft F1 Loss	91.38	11.41	20.16	14.57	5.92	90.72	34.55	18.9	24.43	3.08
	Class Weight	92.16	12.21	18.6	14.74	5.06	89.39	36.97	47.89	41.72	7.04
PolyCoder	No weight	92.73	10.25	12.81	11.39	4.24	91.97	48.76	23.78	31.96	2.15
	Project Balanced	94.37	8.33	5.46	6.59	2.27	90.77	30.08	12.33	17.49	2.47
	Weighted Soft F1 Loss	93.17	11.24	12.69	11.92	3.79	89.88	36.07	35.72	35.9	5.46
	Class Weight	89.76	9.84	22.16	13.63	7.68	86.48	29.19	49.36	36.68	10.32
CodeT5 Small	No Weighting	94.91	13.35	7.24	9.39	1.78	91.85	48.41	42.22	45.10	3.88
	Project Balanced Batch Sampler	95.3	14.52	5.90	8.39	1.31	90.69	39.36	31.96	35.27	4.24
	Weighted Soft F1 Loss	96.34	48.18	5.90	10.52	0.24	91.31	44.69	39.78	42.09	4.24
	Class Weights for Cross Entropy Loss	**93.87**	**16.95**	**17.48**	**17.21**	**3.24**	**89.57**	**39.80**	**61.33**	**48.28**	**7.99**

Table 6: Using class weights for cross entropy loss improves the generalization performance of models, when they are trained on seen projects and tested on unseen projects. Using class weights improves the unseen project test F1 score of CodeBERT from 11.94% to 14.74%, PolyCoder from 11.39% to 13.63%, and CodeT5 Small from 9.39% to 17.21%. Moreover, if the training and testing samples are drawn from the same distribution, using class weights also improves the test F1 score. We highlight the row with the highest F1 score in bold.

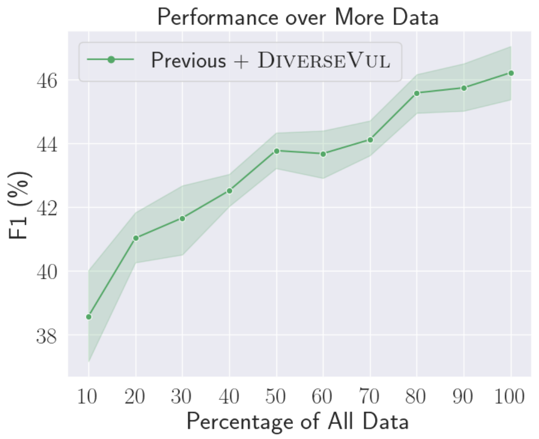

Figure 3: Deep learning for vulnerable source code detection benefits from more data collected from the same distribution as the test data. We fine-tune CodeT5 Small models on different amounts of vulnerable source code data with different volume and report the test F1 score. We run each dataset setup 10 times. The lines are the average, and the region denotes 95% confidence interval. This figure shows that a larger training set improves the F1 score on vulnerability detection on test data from the same distribution.

vulnerable functions a developer is writing, in a new project it has not been trained on. Alternatively, static analyzers can be used to examine vulnerabilities in different projects. In a similar use case, deep learning based detection model needs to analyze a new project (after development) it has not seen before. Both of these use cases require the deep learning model to have strong generalization performance to new projects, and it is an open research problem for the community to tackle.

4.5 Weighting

In this section, we investigate whether three simple weighting schemes can potentially improve the model's generalization performance to unseen test projects. The weighting schemes are the following.

4.5.1 Project Balanced Batch Sampler. Our idea is to make the model perform equally well on different projects. Therefore, we propose a batch sampler that is equally likely to sample from any project in the training set. If a project is picked, it then randomly sample from all functions belonging to the project.

4.5.2 Weighted Soft F1 Loss. Since we care about F1 score as the final performance metric, we would like explore if a different loss function helps with improving the generalization performance. We use normalized prediction probabilities (between 0 and 1) from the training samples to calculate true positives, true negatives, false positives, and false negatives, as in floating point numbers. Then, we use these to compute two F1 scores of predicting the positive label (vulnerable function) and the negative label (nonvulnerable functions) separately. The loss for the positive label is 1 - positive F1 score, and the loss for the negative label is 1 - negative F1 score. Finally, we give a higher weight to the first loss value, proportional to the ratio of nonvulnerable to vulnerable functions in the data. Then, we choose the corresponding loss value according to the ground truth class label as the final training loss.

4.5.3 Class Weights for Cross Entropy Loss. In this scheme, we still use cross entropy loss for training. We upweight the loss value for

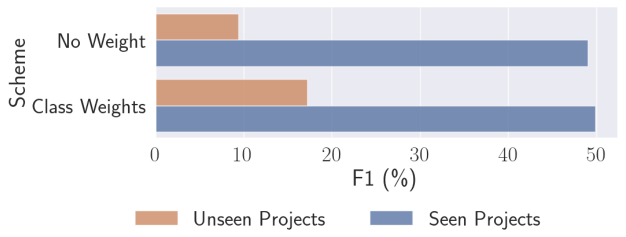

Figure 4: Using class weights in the training loss function improves the generalization performance over unseen projects for CodeT5 Small, and it slightly improves the performance on seen projects as well. The test F1 score on unseen projects is still quite low.

the positive class (vulnerable class), proportional to the ratio of nonvulnerable samples over vulnerable samples. We use the same loss value for the negative class.

4.5.4 Results. We follow the same project split dataset setup described in Section 4.4. We fine tune CodeBERT, PolyCoder, and CodeT5 Small models over the seen projects training set from Previous + DiverseVul dataset, and test them on 95 unseen projects. For each model architecture, we use four schemes to fine tune four models: no weighting, project balanced batch sampler, weighted soft F1 loss, and class weights for cross entropy loss. In addition, we fine tune another four models for each architecture using these schemes over a different data split, the random data split described in Section 4.2.

Result 9: Using class weights for cross entropy loss can improve the model's generalization performance to unseen projects, but there is a lot of room for further improvements. Class weights also improve the model's performance if training / testing samples are drawn from the same distribution. Table 6 shows the evaluation results of models fine tuned with different schemes. For the seen / unseen projects experiment, using class weights increases the F1 score for all three model architectures. The project balanced batch sampler does not help with generalization. The weighted soft F1 loss helps CodeBERT and CodeT5 Small with generalization, but it hurts performance on seen projects. Overall, class weights is the best scheme, as it improves performance on both seen and unseen projects. CodeT5 Small trained with class weights has the best test F1 score (17.21%) on unseen projects.

Figure 4 shows the gap between the F1 score on seen projects vs unseen projects for two CodeT5 Small models, one fine tuned with no weighting scheme and one fine tuned with class weights for cross entropy loss. From the bars, we observe that using class weights reduces the gap between F1 score on seen vs unseen projects, with slight improvement to F1 score on seen projects and significant improvement for unseen projects. This means that using class weights improves the performance of the model over samples drawn from the same distribution as well as from a different distribution of new projects. However, there is still a large gap between 49.9% F1 on seen projects vs 17.21% F1 on unseen projects. As future research directions of the generalization problem, there is a lot of potential to further improve the model's performance over unknown projects.

4.6 Performance on CWEs

To understand the difficulty of learning different CWEs, we select 37 CWEs to examine the CodeT5 Base model's prediction performance when it is trained on Previous + DiverseVul. The 37 CWEs include the top-25 CWEs according to MITRE [6], and the 12 most common CWEs in DiverseVul outside the top 25. We select vulnerable functions belonging to these 37 CWEs and all nonvulnerable functions from the Previous + DiverseVul test set obtained from the random split in Section 4.2.

Result 10: Some CWEs are easier to learn than others regardless of the training data size. Table 7 shows the CodeT5 Base model's prediction performance across the 37 CWEs. We have highlighted the 10 most prevalent CWEs in the training set and 10 highest True Positive Rate (TPR) numbers in bold. Note that all CWEs have the same False Positive Rate (FPR) since FPR is only related to nonvulnerable functions. We observe that having more samples for a particular CWE in the training set does not necessarily result in the model learning it better than CWEs with fewer training samples. Moreover, some CWEs with very few training samples are well-detected by the model. For example, CWE-502, CWE-79, CWE-89, all of which account for less than 2% of the training data, have the highest TPRs. This suggests that some CWEs are easier to learn and do not require a large amount of training data, while others are more challenging to learn, even with more training samples. For instance, CWE-416 had 5.46% of the training samples, but its TPR was only 17.86%.

For some CWEs, we do not have enough test samples, resulting in extremely low TPR numbers. The "Test #" column shows the number of vulnerable functions belonging to that CWE in the test set. For CWEs with 0% TPR, most have less than 10 samples in the test set.

5 LABEL ERROR ANALYSIS

While our dataset is designed to be as accurate as possible, some functions may be labelled erroneously. To label vulnerable functions, we follow the methodology used in Devign [33], ReVeal [4], BigVul [9], CrossVul [19], and CVEFixes [2], which considers a function vulnerable if it was changed by a commit that is identified as fixing a vulnerability, based on security issue trackers. Although our labeling technique is state-of-the-art and can scale effectively, we cannot guarantee that every function changed by each such commit is vulnerable, so some labels may be inaccurate.

To quantify the amount of label noise as a result of this labeling methodology, we manually assess the accuracy of labels for the DiverseVul, CVEFixes, BigVul, and CrossVul datasets. Among previous datasets, we chose CVEFixes, BigVul, and CrossVul because they provide the commit ID that changed the vulnerable function, which allows us to verify whether a function is vulnerable in that specific version of the project.

We randomly sample 50 vulnerable functions from DiverseVul, and 50 vulnerable functions from the union of previous three datasets (CVEFixes ∪ BigVul ∪ CrossVul). Then, we manually analyze whether the vulnerable function has the correct label or wrong label. We inform this decision by examining the code of the function labelled vulnerable, both before and after the commit, the commit it was supposedly fixed in, the CVE description, and developer

CWE	Train (%)	Test #	TPR (%)	FPR (%)	
CWE-119	**15.16**	313	**39.30**	3.55	Improper Restriction of Operations within the Bounds of a Memory Buffer
CWE-120	2.29	49	**40.82**	3.55	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
CWE-125	**11.08**	239	27.20	3.55	Out-of-bounds Read
CWE-189	2.97	57	**31.58**	3.55	Numeric Errors
CWE-190	**4.77**	100	21.00	3.55	Integer Overflow or Wraparound
CWE-200	**5.10**	131	**31.30**	3.55	Exposure of Sensitive Information to an Unauthorized Actor
CWE-20	**10.76**	224	**32.59**	3.55	Improper Input Validation
CWE-22	1.13	20	25.00	3.55	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
CWE-264	3.55	73	28.77	3.55	Permissions, Privileges, and Access Controls
CWE-269	1.14	23	8.70	3.55	Improper Privilege Management
CWE-276	0.19	3	0	3.55	Incorrect Default Permissions
CWE-284	3.35	77	25.97	3.55	Improper Access Control
CWE-287	0.58	10	10.00	3.55	Improper Authentication
CWE-306	0.00	0	N/A	3.55	Missing Authentication for Critical Function
CWE-310	1.95	44	25.00	3.55	Cryptographic Issues
CWE-352	0.10	1	0	3.55	Cross-Site Request Forgery (CSRF)
CWE-362	2.62	61	16.39	3.55	Race Condition
CWE-369	1.26	31	29.03	3.55	Divide By Zero
CWE-399	**5.29**	110	**41.82**	3.55	Resource Management Errors
CWE-400	2.38	34	5.88	3.55	Uncontrolled Resource Consumption
CWE-401	1.83	33	24.24	3.55	Missing Release of Memory after Effective Lifetime
CWE-415	1.55	30	30.00	3.55	Double Free
CWE-416	**5.46**	112	17.86	3.55	Use After Free
CWE-434	0.07	1	0	3.55	Unrestricted Upload of File with Dangerous Type
CWE-476	**5.00**	106	17.92	3.55	NULL Pointer Dereference
CWE-502	0.05	3	**66.67**	3.55	Deserialization of Untrusted Data
CWE-611	0.09	3	0	3.55	Improper Restriction of XML External Entity Reference
CWE-703	**6.39**	133	10.53	3.55	Improper Check or Handling of Exceptional Conditions
CWE-77	0.18	6	16.67	3.55	Command Injection
CWE-78	0.38	7	0	3.55	OS Command Injection
CWE-787	**15.57**	311	**33.76**	3.55	Out-of-bounds Write
CWE-79	0.47	12	**50.00**	3.55	Cross-site Scripting
CWE-798	0.01	0	N/A	3.55	Use of Hard-coded Credentials
CWE-862	0.26	6	16.67	3.55	Missing Authorization
CWE-89	0.31	9	**33.33**	3.55	SQL Injection
CWE-918	0.02	4	0	3.55	Server-Side Request Forgery (SSRF)
CWE-94	0.69	15	0	3.55	Improper Control of Generation of Code ('Code Injection')

Table 7: We evaluate the prediction performance of the CodeT5 Base model across top-25 CWEs and 12 most popular CWEs in DiverseVul. We highlight the 10 highest training sample percentages and 10 highest TPR numbers in bold. Having more training samples for a specific CWE does not necessarily improve the model's prediction performance, and some CWEs are harder to learn than others. Most CWEs with 0% TPR have under 10 samples in the test set.

Dataset	Correct Label	Wrong Label		
		Vulnerability Spread Across Multiple Functions	Relevant Consistency	Irrelevant
DiverseVul	60%	10%	12%	18%
CVEFixes ∪ BigVul ∪ CrossVul	36%	12%	12%	40%
CVEFixes	51.7%	10.3%	17.3%	20.7%
BigVul	25%	15.6%	9.4%	50%
CrossVul	47.8%	13%	21.8%	17.4%

Table 8: Label accuracy of four datasets, evaluated on a random sample of vulnerable functions.

discussions in the security issue tracker. We confirm a function as correctly labelled vulnerable if the vulnerability exists in that function, and is not spread across multiple functions. We observed three categories of label errors: 1) the vulnerability is spread across multiple functions, 2) the function is not vulnerable, but changing the function is relevant to fixing the vulnerability (e.g., to adjust calling parameters), and 3) the function is not vulnerable and irrelevant to the vulnerability (e.g., a vulnerability-fixing commit changes the spaces in some nonvulnerable functions, or makes irrelevant functionality changes to nonvulnerable functions).

Table 8 shows our analysis results. The vulnerable function labels are 60% accurate in DiverseVul, which is 24 percentage points higher than the previous three datasets (CVEFixes ∪ BigVul ∪ CrossVul). Within these three datasets, CVEFixes is the most accurate one, whereas BigVul has very low label accuracy, only 25%. We observe that many commits included in BigVul from the Chromium and Android projects are not relevant to fixing vulnerabilities at all. We also found that the percentage of irrelevant functions is surprisingly high, ranging from 17.4% to 50% in four datasets. These functions are not related to the vulnerability, but since they were changed by the vulnerability-fixing commits, the automatic labeling process labels them as vulnerable.

Concurrent work also examined label noise and also found significant label errors in the BigVul and Devign datasets [7]. Compared to their categorization, we have a stricter criteria to label a function as vulnerable: we consider the caller of a vulnerable function as non-vulnerable; they considered it vulnerable. Also, if a function is only part of the vulnerability, and if the vulnerability cannot be recognized from the code of this function alone, we consider that a wrong label; they considered it correct. Taking into account the differences in categorization, our findings for BigVul (the only dataset common to their and our work) are largely consistent with their findings.

6 LIMITATIONS

The label noise in our dataset and prior datasets may introduce errors into our measurement of the performance of all models on the test set. We hope that releasing our dataset will enable the community to explore methods to remediate the effects of label noise in the future.

In retrospect, the de-duplication procedure in our dataset and prior datasets could be improved. As part of the label noise analysis, we discovered that 4% of DiverseVul labels and 6% of (CVEFixes ∪ BigVul ∪ CrossVul) labels were erroneous because the commit made whitespace-only changes to some functions, and these were treated as security fixes during labelling. Therefore, normalizing the whitespace in all functions before de-duplication could slightly improve label accuracy, and might have other benefits.

There is a risk of contamination, i.e., test data leaking into pretraining data, as LLMs are pre-trained on text and code, which could conceivably include blog articles or code patches related to security vulnerabilities included in our test set. Many of our models (CodeBERT, GraphCodeBERT, PolyCoder, CodeT5 Small, CodeT5 Base, NatGen) were only pre-trained on code, not on other text or code changes, so could have been exposed to code in our test set but were unlikely to be exposed to a description of which code is vulnerable. This could potentially affect our results in ways that we cannot measure. Other models (RoBERTa, GPT-2 Base, CodeGPT, T5 Base) were pre-trained on text, and so could possibly have been exposed to blog articles that describe vulnerable source code. We suspect that this is very rare, but we cannot measure it, so we cannot rule out the possibility of test set contamination. The latter models (RoBERTa, GPT-2 Base, CodeGPT, T5 Base) performed relatively poorly in our experiment in any case.

There is also a risk that cloned code could cause test set contamination, if the cloned code was subsequently modified slightly (thus evading our de-duplication efforts).

7 CONCLUSION

This paper presents a new dataset, DiverseVul, for detecting software vulnerabilities using deep learning. The dataset contains 18,945 vulnerable functions spanning 155 CWEs and 330,492 nonvulnerable functions, extracted from 7,514 commits, which is more diverse and twice the size of the previous largest and most diverse dataset, CVEFixes. We use this new dataset to study the effectiveness of various deep learning architectures in detecting vulnerabilities. We have experimented with 11 different deep learning architectures from four model families: Graph Neural Networks (GNN), RoBERTa, GPT-2, and T5. The results suggest that the increased diversity and volume of training data examined in this paper is beneficial for vulnerability detection, especially for large language models, but it is unclear whether even larger datasets would help or not. Code-specific pretraining tasks appear to be a promising research direction for deep learning based vulnerability detection. Our results highlight a major challenge for future research: improving deep learning models so they generalize to unknown projects. We release the DiverseVul dataset to the community at https://github.com/wagner-group/diversevul.

ACKNOWLEDGMENTS

We are grateful to Bryce Casaje for his contributions exploring multiple approaches for dataset construction, including manual labelling of commits, automated text-based labelling, and more. We are grateful to Kexin Pei for his advice on large language model fine tuning. We gratefully acknowledge the anonymous reviewers for many helpful remarks that significantly improved the paper. This research was supported by the NSF under grant CNS-2154873, by the joint KACST - UC Berkeley Center of Excellence for Secure Computing, by C3.AI's Digital Transformation Institute, and by the Center for AI Safety Compute Cluster. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Seth Hallem Bryan Fulton, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (February 2010).
[2] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. 30–39.
[3] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T Devanbu, and Baishakhi Ray. 2022. NatGen: generative pre-training by "naturalizing"

source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 18–30.

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

[5] Alexis Challande, Robin David, and Guénaël Renault. 2022. Building a Commit-level Dataset of Real-world Vulnerabilities. In *Proceedings of the Twelveth ACM Conference on Data and Application Security and Privacy*. 101–106.

[6] The MITRE Corporation. Last accessed on March 28, 2023. 2022 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

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

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

[9] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. AC/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[11] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[12] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

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

[14] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2021), 2244–2258.

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

[16] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[17] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

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

[19] Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. 2021. CrossVul: a cross-language vulnerability dataset with commit data. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1565–1569.

[20] National Institute of Standards and Technology. Last accessed on March 19, 2023. National Vulnerability Database. https://nvd.nist.gov/

[21] National Institute of Standards and Technology. Last accessed on March 19, 2023. NIST Software Assurance Reference Dataset. https://samate.nist.gov/SARD

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

[23] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[24] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

[25] 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)*. IEEE, 757–762.

[26] 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)*.

[27] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. 2022. Transformer-Based Language Models for Software Vulnerability Detection. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 481–496.

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

[29] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[30] Frank F Xu, Uri Alon, Graham Neubig, and Vincent J Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. *arXiv preprint arXiv:2202.13169* (2022).

[31] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 590–604.

[32] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2A: a dataset built for AI-based vulnerability detection methods using differential analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 111–120.

[33] 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).

A MODEL TRAINING SETUPS

A.1 ReVeal Setup

We use Joern on GitHub [2] to obtain the Code Property Graphs. This is a newer version than what ReVeal used, because if we use the same old version of Joern as in the ReVeal paper, almost half of the functions in all datasets cannot be extracted into graphs.

For the Gated Graph Neural Network, we set maximum training epochs to be 50 for Previous + DiverseVul dataset and 100 for Previous dataset, and pick the model with the best validation F1 score, for experiments in Section 4.2. We set maximum training epochs to be 60 for experiments in Section 4.4. We follow the original setting in ReVeal source code to use Adam optimizer with learning rate 0.0001, and weight decay 0.001.

To train the classification layers in ReVeal, we set the maximum number of epochs to be 100 and follow authors' set up: we stop the training procedure if F1-score on validation set does not increase in 5 epochs. We follow the original setting in ReVeal source code to use Adam optimizer with learning rate 0.001, and no weight decay.

A.2 Fine Tuning Setup

To fine tune LLM models, we apply a linear classification head over the Tranformer model, following standard methods. For RoBERTa, CodeBERT, and GraphCodeBERT, we apply the linear layer over the embedding that represents the first token ([CLS]). For GPT-2-Base, CodeGPT, and PolyCoder, we apply the linear layer over the embedding of the last token. For the T5 Base, CodeT5 Small, CodeT5 Base, and NatGen, we apply the linear layer over the embeddings of the last decoder state.

We use training batch size 32, learning rate 2e-5, Adam optimizer, and train for 10 epochs. We use a linear learning rate decay with warm up of 1,000 steps. We check the model's validation performance every 1,000 steps, and save the model with the best validation performance for testing. We use the same learning rate for all models and all training data setups with one exception. When we train RoBERTa on Previous + DiverseVul from the random data split (in Section 4.2), we use learning rate 1e-5, since a larger learning

[2] After commit a6aa08ee9842eedb52e149695e3a34500b6ceab0 on Oct 11, 2022.

rate results in a degenerate model that always predicts a function as nonvulnerable.