VinJ: An Automated Tool for Large-Scale Software Vulnerability Data Generation

Yu Nong
Washington State University
Pullman, USA
yu.nong@wsu.edu

Haoran Yang
Washington State University
Pullman, USA
haoran.yang2@wsu.edu

Feng Chen
University of Texas at Dallas
Richardson, USA
feng.chen@utdallas.edu

Haipeng Cai
Washington State University
Pullman, USA
haipeng.cai@wsu.edu

ABSTRACT

We present VinJ, an *efficient* automated tool for *large-scale diverse* vulnerability data generation. VinJ automatically generates vulnerability data by injecting vulnerabilities into given programs, based on knowledge learned from existing vulnerability data. VinJ is able to generate diverse vulnerability data covering 18 CWEs with 69% success rate and generate 686k vulnerability samples in 74 hours (i.e., 0.4 seconds per sample), indicating it is *efficient*. The generated data is able to improve existing DL-based vulnerability detection, localization, and repair models significantly. The demo video of VinJ can be found at https://youtu.be/-oKoUqBbxD4. The tool website can be found at https://github.com/NewGillig/VInj. We also release the generated large-scale vulnerability dataset, which can be found at https://zenodo.org/records/10574446.

CCS CONCEPTS

• **Software and its engineering–AI and software engineering**;

KEYWORDS

Vulnerability analysis, data augmentation, deep learning

ACM Reference Format:
Yu Nong, Haoran Yang, Feng Chen, and Haipeng Cai. 2024. VinJ: An Automated Tool for Large-Scale Software Vulnerability Data Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24), July 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3663529.3663800

1 INTRODUCTION

Data-driven techniques have showed great promise over traditional ones [15, 16] for vulnerability analysis. Yet the scarcity of quality data has become the main barrier for further advancing those techniques [3–5, 17, 20]. An intuitive solution is to automatically generate vulnerability data. However, existing automatic vulnerability dataset generation tools have conspicuous limitations. First, some existing tools can only generate vulnerabilities specified by developers. For instance, the framework developed by Zhang [27] can only generate one type of vulnerability, while FixReverter [28] is limited to three manually derived patterns. Learning-based approaches are also used for vulnerability generation. However, they still suffer from major limitations. For example, SemSeed [22] integrates word embedding and pattern mining to inject bugs into programs, but it is only effective for simple bug injection cases (e.g., change <= to <). Getafix [1] uses mined edit patterns to inject vulnerabilities, but lacks sensitivity to code semantics, resulting in noisy generated data. Some DL-based tools (e.g., CodeT5 [24] and Graph2Edit [26]) may be more *semantics-aware*, but they need large amounts of training data which makes them *high-cost*.

To overcome these limitations, we recently developed VulGen [19], a technique that aims at generating quality vulnerability data without large amounts of training data [19]. However, the VulGen prototype only demonstrates its effectiveness for realistic vulnerability data generation without considering the *efficiency*. In the pattern mining phase, all the patterns are clustered without grouping, making the pattern mining very slow. In the data generation phase, it can only generate 17.5 samples per minute on average. This seriously impedes VulGen for *large-scale* vulnerability data generation.

Therefore, based on VulGen, we developed VinJ, an *efficient* automated software vulnerability data generation tool which supports *large-scale* vulnerability data generation. VinJ generates vulnerability samples by injecting vulnerabilities into existing real-world normal programs based on the knowledge learned from existing vulnerability samples. Given a vulnerability training set, VinJ first fine-tunes a semantics-aware large pre-trained programming language model CodeT5 [24] to locate the statements to inject vulnerabilities. Then, VinJ mines the vulnerability-injection patterns from existing vulnerability samples. To overcome the inefficiency of the VulGen prototype, we group the edit patterns based on their root node types before the clustering. Then, we cluster multiple groups at the same time with the support of multi-process parallelization. Finally, given a normal program, VinJ locates a statement with the fine-tuned CodeT5 model and applies a pattern on the located statement to inject a vulnerability. To apply an appropriate pattern, VinJ pre-ranks the patterns rather than ranking them during the generation phase as VulGen does to significantly improve the *efficiency*, making VinJ support *large-scale* vulnerability data generation.

FSE Companion '24, July 15–19, 2024, Porto de Galinhas, Brazil

ACM ISBN 979-8-4007-0658-5/24/07
https://doi.org/10.1145/3663529.3663800

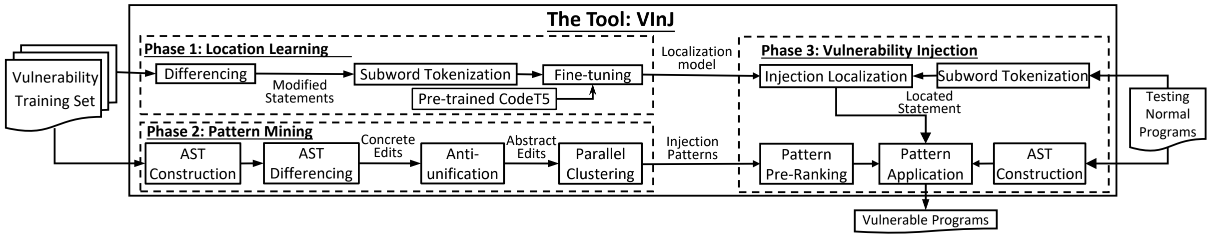

Figure 1: The architecture of VinJ, highlighting its three main working modules (phases).

We have implemented VinJ in Python and trained the VinJ models with 9,705 training samples. When applying on 1,078 testing samples, VinJ can generate *diverse* vulnerability samples covering 18 CWEs. VinJ outperforms the baseline tools and achieves 14.6% accuracy (i.e., percentage of generated samples exactly matching ground truth) and a 69% success rate (i.e., percentage of generated samples that are vulnerable, whether or not matching ground-truth), indicating that VinJ is *effective* for *quality* vulnerability data generation. VinJ is also efficient for large-scale generation: it can generate 686k vulnerability samples in 74 hours (i.e., 0.4 seconds per sample), outperforming VulGen (3.06 seconds per sample). By augmenting their original vulnerability training sets with 10% of the 686k generated samples, state-of-the-art DL-based vulnerability detection, localization, and repair models have their performance improved significantly, indicating that VinJ is practically useful.

The core technical ideas and design rationales underlying VinJ have been published in our research paper on VulGen [19]. Thus, in this demo paper, we focus on **(1)** the additional, important design and implementation details that are not presented in the research paper. Moreover, this tool demo papers include **(2)** tooling enhancement over the original VulGen prototype (e.g., the newly designed performance optimizations such as parallel pattern clustering and pattern pre-ranking), **(3)** extended efficiency and scalability evaluation (e.g., those metrics of the tool in large-scale data generation), and **(4)** much expanded practicability evaluation (e.g., assessing how the data generated by our tool help improve *multiple* downstream vulnerability analysis tasks, including learning-based vulnerability detection, localization, and repair). In contrast, the original research paper only evaluates VulGen in generating a small number of (963) samples from normal programs drawn from a benchmark dataset (rather than from the wild), and in improving the performance of only one downstream task (i.e., learning-based vulnerability detection). Finally, as a tool demo, **(5)** this paper describes how to use our tool (especially via the tool demo walk-through presented in the Appendix), which is also not in the research paper.

To the best of our knowledge, VinJ is the first *efficient* automated tool for *large-scale diverse* vulnerability data generation. Besides, as part of VinJ, we also provide the first open-source implementation of getafix [1], a commercial tool at Meta for bug repair, which is not available before. This is our *another tooling contribution*.

The target audience of VinJ includes any users who need large-scale vulnerability datasets for training and benchmarking data-driven vulnerability analysis models.

2 TOOL DESIGN AND IMPLEMENTATION

Figure 1 shows the architecture of VinJ, highlighting its inputs and outputs as well as three working phases.

Table 1: Hyperparameters of the localization model.

e	number of epochs	10
eb	number of encoder blocks	12
db	number of decoder blocks	12
es	encoder block size	1024
ds	decoder block size	256
b	batch size	1
lr	learning rate	2e-5
bs	beam search size	1

2.1 Inputs and Outputs

To use VinJ, the users need to provide two sets of inputs: a *Vulnerability Training Set* of paired normal and vulnerable programs and *Testing Normal Programs* the users want to inject vulnerabilities.

The outputs of VinJ are the *Vulnerable Programs*.

2.2 Phase 1: Location Learning

During this phase, VinJ learns where to inject (i.e., finding injection locations) from the vulnerability training set with three modules: *Differencing*, *Subword Tokenization*, and *Fine-tuning*.

Differencing. Given the Vulnerability Training Set, VinJ first differences on the pairs of normal programs and respective vulnerable programs. For implementation, we use the `diff` tool in Linux to do this. It compares a pair of programs and outputs the lines changed, which are the *Modified Statements*. In this case, the normal program and the respective modified statements construct a pair of input and output for the *localization model*. These pairs allow VinJ to learn the locations where vulnerabilities may be injected.

Subword Tokenization. To allow the deep learning model to process the pairs, VinJ tokenizes the pairs of normal programs and the respective *modified statements*. It converts the text into a sequence of tokens so that the DL model can process. For implementation, we use the pre-trained CodeT5 Tokenizer from HuggingFace [25] which implements Byte Pair Encoding to do this.

Fine-Tuning. Finally, VinJ fine-tunes the *Pre-trained CodeT5* to train a *Localization Model*. Given the pairs of normal programs and the respective modified statements. VinJ fine-tunes the CodeT5 model so that the model can predict a statement to inject a vulnerability given a new normal program. For implementation, we adapt the source code from VulRepair [11] as it has similar formulation to ours (i.e., transforming a text to another text). We download the Pre-trained CodeT5 from HuggingFace [25] to fine-tune it. We configure the model with the hyperparameters shown in Table 1. Since we only generate one vulnerable program given a testing normal program, we set the beam search size *bs* to be one. We also enable AdamW optimizer [13] which is widely used to make the training stable.

2.3 Phase 2: Pattern Mining

VinJ mines the vulnerability *Injection Patterns* based on Getafix [1]. It also takes the Vulnerability Training Set as the input and learns the edit patterns to convert normal code into vulnerable code.

AST Construction. Given that each vulnerability fix involves a pair of normal and the respective vulnerable program, VinJ first converts the pair of programs into ASTs. For implementation, we use srcML [6] as the AST parser, which supports C language (the language of our samples), to convert the source code into ASTs.

AST Differencing. Then, VinJ differences the pairs of ASTs to get the *Concrete Edits* between normal and vulnerable programs. We use GumTree [7] to get the concrete edits, following the approach in Getafix [1]. Specifically, given the pair of ASTs, GumTree compares the ASTs and finds the modified AST nodes and subtrees. The pairs of modified subtrees constitute the *concrete edits*.

Anti-Unification. Next, VinJ uses the anti-unification algorithm from Getafix [1] to merge similar concrete edits into *Abstract Edits*. For example, two concrete edits `"safe_free(ptr1) =>free(ptr1)"` and `"safe_free(ptr2) => free(ptr2)"` can be merged into `"safe_free(h0) => free(h0)"` where `h0` is a placeholder that can match any AST subtrees. Since Getafix is not open-source, we re-implement the anti-unification algorithm.

Parallel Clustering. Since there are many edits to be merged, VinJ needs to decide which two edits are merged each time. The intuition is that the merging should start from the most similar concrete edits. Thus, we further follow Getafix and re-implement the hierarchical clustering algorithm to merge the abstract edits into generalizable edit patterns. Given a set of abstract edits, VulGen enumerates all possible edit pairs to find the most similar pair. However, if the root nodes of two edits are different, the edits are the most different and thus the merging is meaningless (i.e., the merged pattern can match any code). Therefore, in VinJ, we optimized the hierarchical clustering into *Parallel Clustering*. Given a set of abstract edits, VinJ first clusters the edits based on the AST root node type. Then, each cluster does the hierarchical clustering independently. This not only reduces the time cost of clustering, but also enables multi-process/parallel computing which further improves efficiency. After the clustering, the mined edit patterns are the *Injection Patterns* with which vulnerabilities may be injected in a variety of normal programs.

2.4 Phase 3: Vulnerability Injection

In this phase, VinJ applies the trained *Localization Model* and the mined *Injection Patterns* to inject vulnerabilities into normal programs. In this process, the VinJ input is the*Testing Normal Programs* which are richly available in real-world projects. Through the vulnerability injection process in VinJ, the VinJ output is the *Vulnerable Programs* it generated.

Subword Tokenization. The testing normal programs are again fed into the subword tokenization model which is described in Section 3. For implementation, we again use the CodeT5 Tokenizer downloaded from HuggingFace [25].

AST Construction. To allow the *Injection Patterns* to match and apply on the code, VinJ again parses the testing normal programs into an AST. For implementation, we again use the srcML [6] tool to help accomplish this step.

Injection Localization. The tokenized programs are fed into the localization model and the model outputs the *located statement*, which is the text of a statement for vulnerability injection. For implementation, we use the part of source code from VulRepair [11] that was used for testing (model inference).

Pattern Pre-Ranking. In the VulGen prototype, given a testing sample, to select an appropriate pattern to apply, the patterns are ranked per the product of two scores: (1) *prevalence score*: the proportion of samples in the training set that can be injected vulnerabilities correctly by applying the pattern, assuming it can be applied on the correct location; (2) *specialization score*: the reciprocal of the number of subtrees in the *testing sample* that the pattern can match. However, this is very inefficient because the specialization score needs to be calculated for each testing sample. In VinJ, we optimize the pattern selection. For specialization score, we calculate the average number of subtrees that match the pattern in the *training samples*. This way, the pattern scores can be calculated before vulnerability injection and the patterns can be *pre-ranked*.

Pattern Application. With the located statement, VinJ applies the first injection pattern per the *pre-ranking* that can match that statement. Since the localization model outputs the statement in source code (text) form, we convert each AST subtree back into source code and compare it with the located statement. Once the source code matches, we apply the injection pattern on that AST.

3 EVALUATION

We conduct our experiments on a machine with an AMD Ryzen Threadripper 3970X (3.7GHz) CPU with 32 Cores, an Nvidia GeForce RTX 3090 GPU, and 256GB memory.

3.1 Effectiveness

To evaluate VinJ, we apply it to the existing vulnerability samples with ground truths. We combine 5 different real-world high-quality vulnerability datasets to build a dataset: Devign [30], ReVeal [5], PatchDB [23], BigVul [8], and CVEFixes [2]. We cleaned up the combined dataset by removing overlapped samples, resulting in 27,237 function-level samples. Since there are irrelevant code changes in multi-line commits for vulnerability fixing [19] and more than 40% of the vulnerabilities only need one-statement injection, we only use samples where the edits are on one statement, resulting in 10,783 samples. We split these samples with a ratio of 9:1 for training and testing as in prior work [18], resulting in 9,705 for training and 1,078 for testing.

Figure 2 shows the results of VinJ and the baselines. VinJ achieves 14.64% exactly-match accuracy, outperforming the baselines. However, it is possible that an output program is vulnerable but does not exactly match the ground truth. Thus, we randomly sample 100 of the generated programs and manually check them. Note that the sample size 100 is sizable as peer works also use 100 or less samples for similar-purpose manual validation [12, 29]. We invite three PhD students who have 2-4 years of experience in software engineering and security to label them. The manual labeling of a sample assesses if an exploit can be written to attack it [19]. We follow an inter-rater agreement procedure and qualify the agreement in terms of Cohen's Kappa, which is a standard metric to evaluate the reliability of the assessment by different raters [9]. The Cohen's

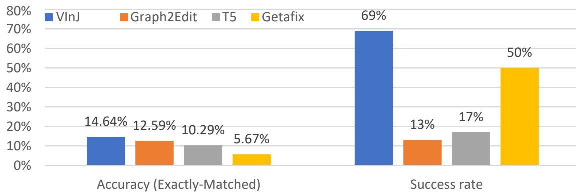

Figure 2: Effectiveness of VinJ and the baselines.

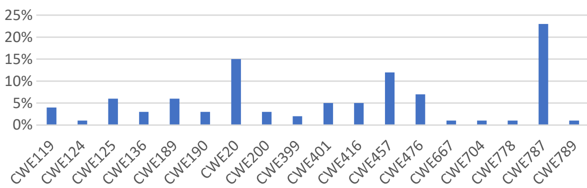

Figure 3: Diversity of VinJ's generated samples.

Kappas between the three participants are 0.7877, 0.7476, and 0.6826, indicating substantial [14] reliability of the manual checking. On average, the three raters label 69% of the generated programs as vulnerable, while the baselines only achieve 13-50% success rate, indicating VinJ is promising to generate vulnerabilities.

We also assess whether VinJ is able to inject a variety of vulnerabilities during the manual labeling. Figure 3 shows the distribution of VinJ's generated data. VinJ covers 18 different CWEs where the distribution is similar to real-world CWE distribution [21], indicating that the generated data is diverse.

3.2 Efficiency and Scalability

We evaluate the efficiency VinJ's major modules (other modules take negligible time and memory). The fine-tuning module can do 8.18 iterations on average, and thus the whole fine-tuning on the 9,705 samples with 10 epochs takes 3 hours 17 minutes, with up to 5.5G memory usage. The AST construction and differencing modules take 1.2 seconds on each sample on average. We enable 60-process configuration and thus the whole AST construction and differencing on the 9,705 samples take only 162 seconds, with up to 73G memory cost (1.21G per process on average). The anti-unification and parallel clustering take 1 hour 17 minutes with up to 22.6G memory cost to cluster these concrete edits with our optimization while the VulGen prototype takes 19 hours 24 minutes with 11.3G memory cost. In the vulnerability injection phase, the pattern pre-ranking takes 16 minutes. After that, the injection localization takes 0.22 seconds for each sample on average with up to 2.2G memory cost, and the pattern application takes 0.18 seconds for each sample on average with up to 13G memory cost. Therefore, VinJ takes 7 minutes 11 seconds to inject vulnerabilities into the 1078 testing samples, while the VulGen prototype takes 55 minutes (i.e., 3.06 seconds per sample), indicating that VinJ is efficient.

We also evaluate the scalability of VinJ for large-scale vulnerability data generation. Thus, we collect a dataset of 738,453 normal programs from 238 open-source projects. After discarding the samples that VinJ cannot inject vulnerabilities, VinJ generates 686,513 samples in 73 hours 14 minutes, with up to 96G memory usage, indicating that VinJ is scalable.

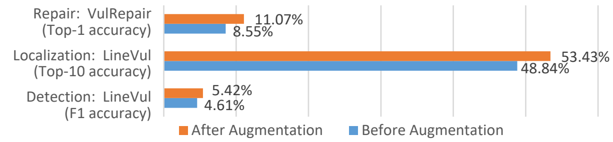

Figure 4: Improvements of downstream tasks with the samples generated by VinJ.

3.3 Practical Usefulness

We practically apply the generated vulnerability samples to improve the training of DL-based downstream vulnerability analysis models. We extensively consider three tasks–vulnerability detection, localization, and repair–while the original VulGen paper focuses solely on detection [19]. For models, we utilize LineVul [10], a vulnerability detection and localization model, and VulRepair [11], a vulnerability repair model. To improve the training of the models, we augment their original training sets by incorporating 10% of the 686,513 generated samples, along with additional normal samples to balance the dataset. The 10% usage is due to scalability considerations for downstream task models. To simulate the real-world vulnerability analysis scenario, we use a third-party testing set ReVeal [5] to leverage independent testing. Figure 4 shows the significant improvements of the three tasks with the generated samples. The default metrics for the three tasks—F1, top-10, and top-1 accuracy—show improvements from 4.61% to 5.42%, 48.84% to 53.43%, and 8.55% to 11.07%, respectively. This shows the practicability of the generated samples.

4 LIMITATION

There are several major limitations of VinJ. The first limitation is that it relies on the AST parser srcML [6]. At the same time, the implementation of VinJ is based on the AST format for C language. Thus, it can only work with C programs with the current implementation. The second limitation is that the injection patterns only work on a single statement. Although this is less restrictive than existing tools that only handle even smaller edits (e.g., token-level [22]), single-statement edits do represent a substantial portion of real-world vulnerability introduction [19]. The third limitation is that, it cannot guarantee the output vulnerability-injected programs are indeed vulnerable, thus the generated data can still suffer from noise (although the noise is less than peer works such as D2A [29], which has only 53% success rate while ours achieves 69%).

5 CONCLUSION

We have developed VinJ, an automated tool that generates vulnerable code by injecting vulnerabilities into existing real-world normal programs, which are richly available. VinJ effectively uses existing high-quality vulnerability samples to train effective vulnerability injection models. VinJ is able to generate large-scale and quality vulnerability data. Our empirical experiments show that VinJ is effective in vulnerability injection and outperforms baseline approaches. It is also efficient to generate large-scale vulnerability datasets which are practically useful.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27. https://doi.org/10.1145/3360585

[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 (PROMISE)*. 30–39. https://doi.org/10.1145/3475960.3475985

[3] Yingzhou Bi, Jiangtao Huang, Penghui Liu, and Lianmei Wang. 2023. Benchmarking Software Vulnerability Detection Techniques: A Survey. *arXiv preprint arXiv:2303.16362* (2023). https://doi.org/10.48550/arXiv.2303.16362

[4] Haipeng Cai, Yu Nong, Yuzhe Ou, and Feng Chen. 2023. Generating vulnerable code via learning-based program transformations. In *AI Embedded Assurance for Cyber Systems*. Springer, 123–138. https://doi.org/10.1007/978-3-031-42637-7_7

[5] 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

[6] Michael L Collard, Michael John Decker, and Jonathan I Maletic. 2013. srcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *2013 IEEE International Conference on Software Maintenance*. 516–519. https://doi.org/10.1109/ICSM.2013.85

[7] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (ASE)*. 313–324. https://doi.org/10.1145/2642937.2642982

[8] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*. 508–512. https://doi.org/10.1145/3379597.3387501

[9] Joseph L Fleiss, Bruce Levin, Myunghee Cho Paik, et al. 1981. The measurement of interrater agreement. *Statistical methods for rates and proportions* 2, 212-236 (1981), 22–23. https://doi.org/10.1002/0471445428.ch18

[10] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: a transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*. 608–620. https://doi.org/10.1145/3524842.3528452

[11] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 935–947. https://doi.org/10.1145/3540250.3549098

[12] Xinlei He, Savvas Zannettou, Yun Shen, and Yang Zhang. 2023. You Only Prompt Once: On the Capabilities of Prompt Learning on Large Language Models to Tackle Toxic Content. In *2024 IEEE Symposium on Security and Privacy (SP)*. 61–61. https://doi.org/10.1109/SP54263.2024.00061

[13] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014). https://doi.org/0.48550/arXiv.1412.6980

[14] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica* 22, 3 (2012), 276–282. https://doi.org/10.23641/asha.12978197

[15] Yu Nong and Haipeng Cai. 2020. A preliminary study on open-source memory vulnerability detectors. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 557–561. https://doi.org/10.1109/SANER48275.2020.9054851

[16] Yu Nong, Haipeng Cai, Pengfei Ye, Li Li, and Feng Chen. 2021. Evaluating and comparing memory error vulnerability detectors. *Information and Software Technology* 137 (2021), 106614. https://doi.org/10.1016/j.infsof.2021.106614

[17] Yu Nong, Richard Fang, Guangbei Yi, Kunsong Zhao, Xiapu Luo, Feng Chen, and Haipeng Cai. 2024. VGX: Large-scale sample generation for boosting learning-based software vulnerability analyses. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. Article 149, 13 pages. https://doi.org/10.1145/3597503.3639116

[18] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2022. Generating realistic vulnerabilities via neural code editing: an empirical study. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1097–1109. https://doi.org/10.1145/3540250.3549128

[19] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2023. VulGen: Realistic vulnerability generation via pattern mining and deep learning. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 2527–2539. https://doi.org/10.1109/ICSE48619.2023.00211

[20] Yu Nong, Rainy Sharma, Abdelwahab Hamou-Lhadj, Xiapu Luo, and Haipeng Cai. 2022. Open science in software engineering: A study on deep learning-based vulnerability detection. *IEEE Transactions on Software Engineering* 49 (2022), 1983–2005. https://doi.org/10.1109/TSE.2022.3207149

[21] National Institute of Standards and Technology (NIST). 2022. CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html.

[22] Jibesh Patra and Michael Pradel. 2021. Semantic bug seeding: a learning-based approach for creating realistic bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 906–918. https://doi.org/10.1145/3468264.3468623

[23] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. 2021. Patchdb: A large-scale security patch dataset. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 149–160. https://doi.org/10.1109/DSN48987.2021.00030

[24] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[25] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771* (2019). https://doi.org/10.48550/arXiv.1910.03771

[26] Ziyu Yao, Frank F. Xu, Pengcheng Yin, Huan Sun, and Graham Neubig. 2021. Learning Structural Edits via Incremental Tree Transformations. In *International Conference on Learning Representations*. 126–133. https://doi.org/10.48550/arXiv.2101.12087

[27] Shasha Zhang. 2021. A Framework of Vulnerable Code Dataset Generation by Open-Source Injection. In *2021 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*. 1099–1103. https://doi.org/10.1109/ICAICA52286.2021.9497888

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

[29] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2A: A dataset built for ai-based vulnerability detection methods using differential analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 111–120. https://doi.org/10.1109/ICSE-SEIP52600.2021.00020

[30] 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 (NeurIPS)* 32 (2019), 10197–10207. https://doi.org/10.48550/arXiv.1909.03496

APPENDIX: VINJ DEMO WALK-THROUGH

In this section, we walk through the demo process of VInJ and use illustrative screen-shots to show the usage of VInJ.

1. INSTALL VINJ

To install VInJ, we first set up the environments and install dependencies that VInJ needs. Then, we clone the VInJ repository to start the experiments.

- Step 1. Download and install Java>=11 through this link.
- Step 2. Download and install srcML through this link.
- Step 3. Download and extract our specific gumtree-lite.zip from figshare.
- Step 4. Set the environment path for our specific gumtree-lite:

```
export GUMTREE=YOUR_GUMTREE_PATH
export PATH=$PATH:$GUMTREE
```

- Step 5. Download and install the Python>=3.6 through this link.
- Step 6. Install the following Python dependencies for the localization model:

```
pip install transformers
pip install torch
pip install numpy
pip install tqdm
pip install pandas
pip install tokenizers
pip install datasets
pip install gdown
pip install tensorboard
pip install scikit-learn
```

- Step 7. Download and install PyPy3 through this link to improve the efficiency.
- Step 8. Clone or download the VInJ repository to the local directory.

2. USE VINJ

In this section, we demonstrate the usage of VInJ for vulnerability dataset generation. Because of the repository size limit on GitHub, we only include a small set of training and testing samples in the repository. The training and testing experiments on the small dataset can be easily run by the scripts (i.e., the .sh files in the repository) provided. We first train and test the localization model with the following script:

```
source loc_train_test.sh
```

Figure 5 shows the run-time log of the localization model and testing. The localization model correctly predicts 25.8% of the vulnerability injection locations.

Then, we extract the injection patterns with the following script:

```
source pattern_train_demo.sh
```

Figure 6 shows the run-time log of pattern extraction. We can see that VInJ extracts and clusters many injection patterns.

```
root@8658d39c86d9:~/test_vinj/VInj# source loc_train_test_demo.sh
05/11/2023 09:22:57 - WARNING - __main__ -   device: cuda:0, n_gpu: 1
05/11/2023 09:23:01 - INFO - __main__ -   Training/evaluation parameters
Namespace(adam_epsilon=1e-08, checkpoint_model_name='non_domain_model.bin',
config_name='', decoder_block_size=512, device=device(type='cuda', index=0),
do_test=False, do_train=True, encoder_block_size=512, epochs=10, eval_batch_size=1,
evaluate_during_training=True, gradient_accumulation_steps=1, learning_rate=2e-05,
max_grad_norm=1.0, max_steps=-1, model_name='model.bin',
model_name_or_path='Salesforce/codet5-base', model_type='t5', n_gpu=1, num_beams=50,
output_dir='./saved_models', seed=123456, tokenizer_name='Salesforce/codet5-base',
train_batch_size=1, warmup_steps=0, weight_decay=0.0)
100%|██████████| 236/236 [00:01<00:00, 221.23it/s]
100%|██████████| 34/34 [00:00<00:00, 160.74it/s]
/opt/conda/lib/python3.7/site-packages/transformers/optimization.py:309:
FutureWarning: This implementation of AdamW is deprecated and will be removed in a
future version. Use the PyTorch implementation torch.optim.AdamW instead, or set
`no_deprecation_warning=True` to disable this warning
  FutureWarning,
05/11/2023 09:23:03 - INFO - __main__ -   ***** Running training *****
05/11/2023 09:23:03 - INFO - __main__ -     Num examples = 236
05/11/2023 09:23:03 - INFO - __main__ -     Num Epochs = 10
05/11/2023 09:23:03 - INFO - __main__ -     Instantaneous batch size per GPU = 1
05/11/2023 09:23:03 - INFO - __main__ -     Total train batch size = 1
05/11/2023 09:23:03 - INFO - __main__ -     Gradient Accumulation steps = 1
05/11/2023 09:23:03 - INFO - __main__ -     Total optimization steps = 2360
epoch 0 loss 4439.68945:   0%|          | 0/236 [00:00<?, ?it/s]
vulrepair_main.py:172: RuntimeWarning: overflow encountered in exp
  avg_loss = round(np.exp((tr_loss - logging_loss) /(global_step- tr_nb)),4)
epoch 0 loss 3484.5952: 100%|█████████| 235/236 [00:29<00:00,  8.13it/s]
05/11/2023 09:23:32 - INFO - __main__ -   ***** Running evaluation *****
05/11/2023 09:23:32 - INFO - __main__ -     Num examples = 34
05/11/2023 09:23:32 - INFO - __main__ -     Batch size = 1
100%|██████████| 34/34 [00:00<00:00, 34.29it/s]
05/11/2023 09:23:33 - INFO - __main__ -   ***** Eval results *****
05/11/2023 09:23:33 - INFO - __main__ -   Evaluation Loss: 1660.26922
epoch 0 loss 3484.5952:  100%|██████████| 236/236 [00:30<00:00,  7.73it/s]
epoch 1 loss 1278.30948: 100%|██████████| 235/236 [00:29<00:00,  8.05it/s]
......
05/11/2023 09:37:39 - INFO - __main__ - ***** Running training *****
05/11/2023 09:37:39 - INFO - __main__ -  Num examples = 31
05/11/2023 09:37:39 - INFO - __main__ -  Batch size = 1
100%|█████████| 31/31 [1:19<00:00,  2.56S/it]
05/11/2023 09:37:39 - INFO - __main__ - ***** Test results *****
05/11/2023 09:37:39 - INFO - __main__ - Test Accuracy: 25.8
```

Figure 5: The run-time log of the localization model training and testing.

```
root@8658d39c86d9:/VInj# source pattern_train_demo.sh
0 sample_388
patterns extracted: 13
1 sample_505
patterns extracted: 13
2 sample_694
patteerns extracted: 6
......
block_content 218
0.08575153350830078
0.0004799365997314453
block_content 218
0.03480410575866699
0.0003669261932373047
block_content 218
0.03881025314331055
0.007937908172607422
block_content 218
0.08478951454162598
0.0004489421844482422
......
```

Figure 6: The run-time log of the pattern extraction.

Finally, we test the extracted patterns and the trained localization model on the testing set using the following script:

```
source pattern_test_demo.sh
```

Figure 7 shows the run-time log of pattern extraction. We can see that VInJ uses the extracted patterns to inject vulnerabilities into the programs.

```
root@8658d39c86d9:/VInj# source pattern_test_demo.sh
sample_1252
fail
del: true
add: true

1 5 3
sample_1444
fail
del: true
add: true

6 1 0
sample_1433
fail
del: true
add: true
```

Figure 7: The run-time log of the vulnerability injection.

To do complete training and testing, we can download the full training and testing datasets for localization and pattern mining/application through the following line. Then replace the respective folders and files with the demo ones.

To make the process simpler, we can also directly download the trained localization model and patterns for testing, by download them in the following link. We can also directly run the script to automatically download models and inject vulnerabilities on our testing set:

```
source pattern_test.sh
```

After this process, the vulnerability-injected samples are outputted to `./pattern_mining_application/generated/`.

Received 2024-01-29; accepted 2024-04-15