applied sciences

MDPI

Article

Improving VulRepair's Perfect Prediction by Leveraging the LION Optimizer

Brian Kishiyama *,† **, Young Lee** † **and Jeong Yang** †

Department of Computational, Engineering, and Mathematical Sciences, Texas A&M University—San Antonio, One University Way, San Antonio, TX 78224, USA; young.lee@tamusa.edu (Y.L.); jeong.yang@tamusa.edu (J.Y.)

* Correspondence: bkishiyama@tamusa.edu

† These authors contributed equally to this work.

Abstract: In current software applications, numerous vulnerabilities may be present. Attackers attempt to exploit these vulnerabilities, leading to security breaches, unauthorized entry, data theft, or the incapacitation of computer systems. Instead of addressing software or hardware vulnerabilities at a later stage, it is better to address them immediately or during the development phase. Tools such as AIBugHunter provide solutions designed to tackle software issues by predicting, categorizing, and fixing coding vulnerabilities. Essentially, developers can see where their code is susceptible to attacks and obtain details about the nature and severity of these vulnerabilities. AIBugHunter incorporates VulRepair to detect and repair vulnerabilities. VulRepair currently predicts patches for vulnerable functions at 44%. To be truly effective, this number needs to be increased. This study examines VulRepair to see whether the 44% perfect prediction can be increased. VulRepair is based on T5 and uses both natural language and programming languages during its pretraining phase, along with byte pair encoding. T5 is a text-to-text transfer transformer model with an encoder and decoder as part of its neural network. It outperforms other models such as VRepair and CodeBERT. However, the hyperparameters may not be optimized due to the development of new optimizers. We reviewed a deep neural network (DNN) optimizer developed by Google in 2023. This optimizer, the Evolved Sign Momentum (LION), is available in PyTorch. We applied LION to VulRepair and tested its influence on the hyperparameters. After adjusting the hyperparameters, we obtained a 56% perfect prediction, which exceeds the value of the VulRepair report of 44%. This means that VulRepair can repair more vulnerabilities and avoid more attacks. As far as we know, our approach utilizing an alternative to AdamW, the standard optimizer, has not been previously applied to enhance VulRepair and similar models.

Keywords: VulRepair; T5 transformer; LION optimizer; AdamW; software vulnerabilities

Citation: Kishiyama, B.; Lee, Y.; Yang, J. Improving VulRepair's Perfect Prediction by Leveraging the LION Optimizer. *Appl. Sci.* **2024**, *14*, 5750. https://doi.org/10.3390/app14135750

Academic Editor: Paolino Di Felice

Received: 8 June 2024
Revised: 25 June 2024
Accepted: 26 June 2024
Published: 1 July 2024

1. Introduction

Software vulnerabilities refer to defects in programming codes or applications that could be exploited by attackers. These vulnerabilities allow attackers to gain access to or damage a computer system. It is apparent that they need to be addressed. The importance of this research is to evaluate and improve tools, specifically AIBugHunter, as it is used to address attacks. AIBugHunter is a tool that scans C/C++ programs, finds code that is vulnerable or susceptible to attacks, identifies the CWE (Common Weakness Enumeration) type of vulnerability, estimates the CVSS (Common Vulnerability Scoring System) vulnerable severity score, explains the vulnerability, and suggests the proper repair. It was developed by the ASWM research group, as reported by Fu et al. (accessed on 2 February 2024 and available at https://github.com/awsm-research/VulRepair/)—who stated that most vulnerabilities are found in C compared to other programming languages [1]. Although there are many Integrated Development Environments (IDEs), the tool can be found and installed in Visual Studio Code as an extension.

A Common Weakness Enumeration, or CWE, is a list of common software and hardware weaknesses that address security concerns. A weakness is a condition in a computer system, including software, hardware, etc., that could cause a vulnerability and should not be used [2]. Regarding software weaknesses, developers cause them when writing programs in specific ways that could be exploited. When creating a CWE, a unique number is assigned to it, referred to as a CWE ID. A created CWE also has a name that describes the weakness, a summary of the weakness, an extended description, the modes of introduction, potential mitigations, common consequences, applicable platforms, demonstrative examples, observed examples, relationships, and references. CWE is a method software developers can use to become aware of and eliminate vulnerabilities or security flaws in their programs. In general, it describes the weakness and provides guidance on how it can be avoided. If a vulnerable function is found, AIBugHunter predicts the CWE-ID, the CWE type, and the severity of the vulnerable function via the Common Vulnerability Scoring System (CVSS) severity score.

The CVSS serves as an industry standard for assessing the impact of vulnerabilities in computer systems. It is beneficial as it offers insights and prompts the necessary responses. The scoring system ranges from 0 to 10, where a score of 10 indicates extreme severity that requires urgent action. The CVSS v3.x ratings are categorized as follows: 0, which means no severity; 0.1 to 3.9, which means low severity; 4 to 6.9, which means medium severity; 7 to 8.9, which means high severity; and 9 to 10, which is critical. More information is available at the National Vulnerabilities Database (NVD) website [3].

On the back end, AIBugHunter integrates LineVul and VulRepair to locate and repair vulnerabilities. LineVul is a transformer-based approach that predicts vulnerable functions and identifies vulnerable lines within programs written in C/C++ code. It addresses the limitations of IVDetect by Li et al [4]. IVDetect is an AI vulnerability detector that uses an intelligence assistant and outperforms 2021 deep learning models [5].

VulRepair, also integrated into AIBugHunter, relies on a T5 architecture. T5 is a natural language processing (NLP) model developed by Google in 2019 [6]. T5 is short for text-to-text transfer transformer with layers that consist of an encoder and decoder neural network. The encoder is used for the input sequence, while the decoder is used for the output sequence. VulRepair's T5 makes use of both natural and programming languages and byte pair encoding (BPE), outperforming other models such as VRepair and CodeBERT. It is used to repair the vulnerability of the function, or, as they say, to suggest repairs to vulnerable lines [7].

Perfect prediction refers to an outcome that is 100% accurate and not a random guess. The perfect prediction percentage provides the percentage of correct outcomes. Regarding VulRepair, it indicates the percentage of cases where a correct repair is matched, or perfectly predicted, to a vulnerable function.

In their study, Fu et al. showed how VulRepair, with a 44% perfect prediction in fixing vulnerabilities, outperforms CodeBERT at 31% and VRepair at 23%. They further demonstrated the importance of using pretraining with natural and programming languages and how a subword tokenizer is better than a word-level tokenizer in the VulRepair model.

Fu et al. used the Big-Vul dataset, which is a C/C++ code vulnerability dataset, to evaluate vulnerable functions [7]. The dataset plays an important role in vulnerability research and was introduced by Fan et al. [8]. It contains 3754 code vulnerabilities and includes CVE IDs and CVE severity scores. Its use applies to file, function, and line fixes. More information is available on GitHub at [9]. Fu et al. also used the CVEfixes dataset, which is used in software security research for vulnerability prediction, classification, severity prediction, and automated vulnerability repair, as provided by Bhandari et al. [10]. The initial release of CVEfixes contains 5495 vulnerability-fixing commits. VulRepair was evaluated on both Big-Vul and CVEfixes, which together contain 8482 vulnerability repairs across over 180 different CWE types.

From [7], we note several points. First, VulRepair achieves optimal performance when the vulnerable functions contain fewer than 500 tokens and the repair tokens are

fewer than 20. The T5 architecture has a token limit of 512, with anything more becoming truncated. Additionally, VulRepair is more effective when vulnerable repairs have fewer than 10 tokens. Second, the provided settings are intended to replicate their experiment, and researchers are encouraged to find methods to deal with larger functions and repair tokens. Third, they discuss threats to their validity, one of which is hyperparameter optimization, which involves finding the optimal hyperparameters for the best results. However, determining the best hyperparameters is costly and time-consuming, as the models require hours for training and testing. Our study asserts that research is continually evolving with ongoing advances. Computer systems, including software, hardware, and algorithms, are constantly improving, which requires continuous testing to evaluate model performance. We address their validity threats and utilize updated libraries to replicate their research. For a comparison of software versions from 2022 and 2024, see Table 1.

Table 1. A comparison of VulRepair's evolving software and hardware.

VulRepair Training to Be Updated		
Library and Other	**2022**	**April 2024**
transformers	4.19.1	4.40.0
torch	1.10.2 + cu113	2.2.1 + cu121
numpy	1.22.3	1.25.2
tqdm	4.62.3	4.66.2
pandas	1.4.1	2.0.3
tokenizers	0.11.6	0.19.1
datasets	2.0.0	2.18.0
gdown	4.5.1	5.1.0
Python	3.9.7	3.10.12
Optimizer	AdamW	optim.torch.AdamW
GPU	NVIDIA 3090	NVIDIA 4090

We also use a 2023 optimizer to help improve the accuracy of their results. Furthermore, Fu et al. reported that the goal of their research was not to find the optimal hyperparameter settings but to compare their model to other models. They suggested that the model could be improved by optimizing the hyperparameters [7]. We also note that the MickeyMike tokenizer is not working. We switch to SalesForce.

The results of our experiment will address the following research question:

> Can we improve the deep learning model's training, enhance vulnerable function detection, and increase the perfect prediction by adjusting VulRepair's hyperparameters and using its current libraries?

2. Related Works

Recent reports indicate that computers and artificial intelligence are rapidly becoming an integral part of our society. They are widely found in various sectors and impact our daily lives. However, the concern with technological advancement is privacy and the potential for information theft. It seems that criminals or criminal groups may exploit computer systems and create chaos. It is not uncommon to hear that trillions of dollars per year are lost due to cybercrimes. Some of these crimes involve taking advantage of software vulnerabilities, which, in turn, becomes an issue for software engineers.

Artificial intelligence has permeated our everyday existence, extending to software engineering. With continuous progress in artificial intelligence and natural language processing, as well as advancements in large language models and multimodal learning, software engineers are incorporating AI techniques throughout the software development

cycle [11]. In the context of identifying vulnerabilities in software code, deep learning methods can aid in detecting these flaws. Ongoing research explores the use of machine learning techniques to discover vulnerability patterns for automated detection, contrasting rule-based methods, and more research is needed to improve them [12].

Deep learning models have the ability to identify new and unseen vulnerabilities by training on similar vulnerabilities. There are limitations since not all vulnerabilities can be detected. Human testers are still needed, as there is still a gap between deep learning techniques and the specialized ability of human code inspectors [13]. Research on automatic software repair to detect vulnerabilities is valuable and a challenging endeavor. The vulnerable code needs to be detected or predicted, and a repair needs to be generated [14].

Gupta et al. used deep learning to fix common errors in the C language as they arise, especially those caused by inexperienced programmers [15]. Feng et al. proposed CodeBERT, a bimodal pretrained in natural language (NL) and programming language (PL), i.e., Python, Java, Ruby, Go, JavaScript, and PHP. CodeBERT learns the semantics between natural and programming languages and is effective in code searches and code documentation generation [16]. Mashadi et al. described debugging as a "time-consuming and labor-intensive task in software engineering" and proposed CodeBERT for automated program repair to fix Java bugs [17]. Marjanov et al. found several studies involving machine learning techniques used in bug detection, such as Google's Error Prone and Spot Bugs. Bug detection and bug correction are a growing field. Most studies target C/C++ and Java, while fewer target Python and other languages. Furthermore, most studies are directed toward semantic defects and vulnerabilities and fewer toward syntactic defects. Moreover, detection and correction studies usually use RNNs along with long short-term memory. However, some also use convolutional neural networks [18]. Fu et al. stated that large language models such as ChatGPT have advanced software engineering tasks related to code review and code generation. However, they found that other models, such as their AIBugHunter and CodeBERT, perform better [19].

We recognize the extensive and intricate domains of security and artificial intelligence models. Our research is confined to static code analysis, which focuses on detecting vulnerabilities in software code before deployment. Unlike dynamic code analysis methods that depend on real-time evaluation or a simulated environment, static analysis does not necessitate the execution of the program. Consequently, the software code can be assessed instantly and does not need to be in a fully executable state toward the end of development.

Although advanced tools are used to uncover vulnerabilities, they are often insufficient as they cannot identify all potential issues. Consequently, static code analysis serves only as an aid in detection and cannot be used as a comprehensive solution to identify all vulnerabilities in a program [20]. Finally, depending on the situation, a hybrid approach that combines both static and dynamic methods should be considered and may prove more effective [21].

We specifically examined and reviewed the work of Mike Fu, as referenced earlier [1,4]. To find additional research related to VulRepair, we conducted searches on the Internet and Google Scholar, yielding over 100 articles. Regarding general coding and deep learning principles, besides the VulRepair code from the AWSM research group [22], we also consulted various sources, including [23–27]. Overall, static code analysis is vital as it addresses software vulnerabilities early, and we should continuously advance the software development field by improving the VulRepair model.

3. Methodology

The purpose of this study was to examine VulRepair and improve the model. We gained insights by reviewing AIBugHunter [1], LineVul [4], and VulRepair [7]. In the VulRepair project, there are threats to internal validity. VulRepair outperforms other models such as VRepair and CodeBERT. Although it outperforms the leading models, the goal was not to ensure the use of optimal hyperparameters. In fact, Fu et al. provided hyperparameter settings for replication, which can be used to extend their study. In

addition, Fu et al. pointed out a limitation of the T5 architecture, where the vulnerable function is limited to 512 tokens. So, VulRepair is more accurate when vulnerable functions have fewer than 500 tokens. The performance also decreases with repairs that require more than 10 tokens. More research is needed to improve these limitations [7].

After understanding the areas of VulRepair that needed improvement, we downloaded VulRepair from GitHub [22]. This provided the code to replicate Fu's experiment [7]. We downloaded the code to Google Colab. We upgraded to Colab Pro+, which facilitates long training and testing sessions. Google Drive is attached to Colab. We increased the memory to 100 GB to ensure that we had ample memory to hold the VulRepair files. In addition, we used the Google Chrome browser. When using Colab, we used Google Colab V100 GPU, a high-performance graphics processing unit renowned for its suitability for deep learning and its ability to handle tasks that require substantial memory and processing power [28].

When running the models, we looked for deprecated warnings and fixed them. For example, AdamW is deprecated. We looked up fixes from Hugging Face and Stack Overflow. They essentially recommended replacing AdamW with PyTorch AdamW, as explained later. We replicated the experiments by Fu et al. and gained insight into their models [7]. Next, we looked at the optimizer. Although AdamW is a popular optimizer, we searched for recently developed optimizers suitable for the VulRepair platform. After researching the LION optimizer, we determined that it would be a good fit for the T5 models. We then fine-tuned the hyperparameters by selecting values around their default settings. Due to time and cost constraints, we did not exhaustively test all hyperparameter values. We tested configurations that best minimized the loss function, thus optimizing the model's performance. Our experiments allowed us to improve VulRepair.

3.1. VulRepair Replication

We replicated VulRepair with a 44.9% perfect prediction. This is 1.15% higher than Fu et al.'s 43.75%. However, we did not use the same "MickyMike" tokenizer or model as Fu et al. [29]. We used SalesForce/codet5-base [30]. We also used the current libraries rather than the recommended libraries used two years ago.

We acknowledge that other studies, like ours, may refer to VulRepair with a 44.9% PP, as in a research article from 2024 at [31].

Model 1 (M1) was designated and referred to as VulRepair, as we compared it to variations of VulRepair and other models, that is, CodeBERT and VRepair. The M1, or VulRepair, components were examined: byte pair encoding tokenizer, programming language (PL) pretraining, natural language (NL) pretraining, and the T5 architecture.

In addition, the initial parameters set for VulRepair were the following:

- Tokenizer name = MickyMike/VulRepair;
  - Ours: SalesForce/codet5-base;
- Model name/path = MickyMike/VulRepair;
  - Ours: SalesForce/codet5-base;
- Epochs = 75;
- Encoder block size = 512;
- Decoder block size = 256;
- Train batch size = 4;
- Eval. batch size = 4;
- Test batch size = 1;
- Optimizer = AdamW;
- Learning rate = 2×10^{-5}.

Ten models were made available for testing, including VulRepair, VRepair, and CodeBERT, each undergoing various modifications. We duplicated each model to assess their percentage, as further described (see Figure 1).

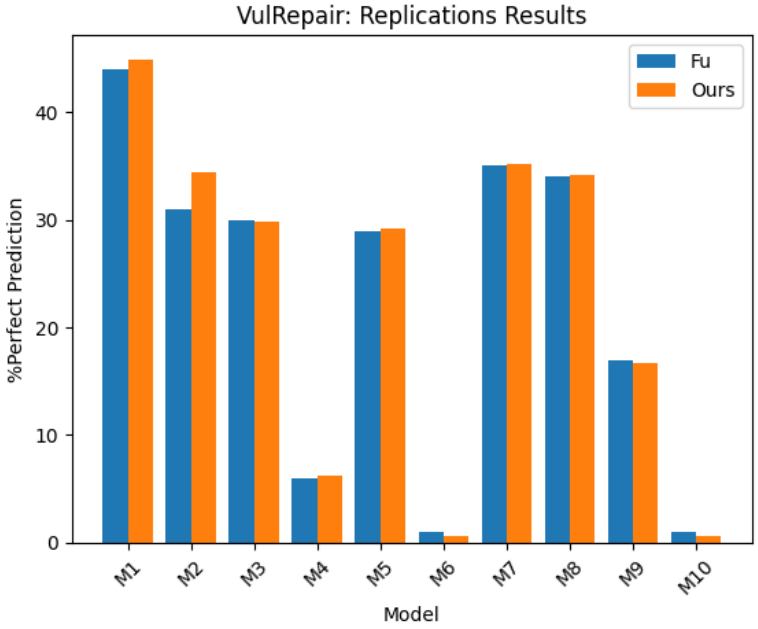

Figure 1. We successfully replicated VulRepair's 44% PP (M1) and other models, as explained in Section 3.2.

3.2. Replication of Other Models

Following the replication of M1, we proceeded with the replication of the models in [7]. Models 1 and 2 were trained from scratch, allowing us to closely examine their training processes. For Models 3 to 10, we utilized the pretrained models supplied by Fu et al. The goal was not to train, enhance, or benchmark CodeBERT and VRepair, but rather to gain a comprehensive understanding of Fu et al.'s research.

Model 2 (M2) is the CodeBERT platform that utilizes a BPE tokenizer, programming language (PL) pretraining, natural language (NL) pretraining, and a BERT architecture. In this case, we trained the model using current libraries and used SalesForce/codet5-base, as in M1, and Google Colab. We also used the deprecated AdamW optimizer, as our intention was not to improve these models. We did not use the Fu et al. 'MikeyMike' tokenizer. We obtained a 34.47% perfect prediction after training and testing. Fu et al. reported a 31% PP [22]. Our results were better by 3.5%, but we did not use the exact methods of Fu et al. We did not conduct further testing to pinpoint the improvements, nor did we use their trained model.

Model 3 (M3) is VulRepair without PL pretraining compared to M1, which includes it. This model demonstrates the importance of PL pretraining. In this instance, we did not train the model but used the one provided by Fu et al. We tested the model and obtained a 29.89% PP. We compared this to Fu et al.'s 30% PP.

Model 4 (M4) is VulRepair with no PL or NL pretraining. This model demonstrates that pretraining yields a better PP. We did not train this model from scratch but tested it. We obtained a 6.21% PP compared to 6% in the Fu et al. model [22].

Model 5 (M5) is CodeBERT, or M2, without PL. This was used to compare VulRepair and CodeBERT when no PL pretraining was used. We tested the model but did not train it from scratch. We obtained a 29.19% PP compared to Fu et al.'s reported results of 29% [22].

Model 6 (M6) is CodeBERT without pretraining. Like M5, it was used to compare with VulRepair's results. Here, we tested it but did not train it from scratch, as this was not our primary focus. We obtained a 0.64% PP compared to Fu et al.'s reported results of 1%.

Model 7 (M7) is VulRepair with a word-level tokenizer that replaces its subword tokenizer. This demonstrates the importance of the tokenizer. We replicated this through testing. Since it was not our focus, we did not train it from scratch. We obtained a 35.17% PP as expected, consistent with the Fu et al. model's results of 35%.

Model 8 (M8) is VRepair, or the vanilla transformer, with a subword tokenizer instead of a word-level tokenizer. This model was compared to both VulRepair and CodeBERT. We tested this model but did not train it from scratch, obtaining a 34.17% PP as expected, consistent with Fu et al.'s results of 34% [22].

Model 9 (M9) is CodeBERT with a word-level tokenizer instead of a subword tokenizer. This was compared to VulRepair's results, demonstrating the importance of tokenization. We tested the model but did not train it from scratch. We obtained a 16.65% PP compared to 17% in the Fu et al. model [22].

Model 10 (M10) is VulRepair with no pretraining and a word-level tokenizer instead of a subword tokenizer. This shows that VulRepair requires NL and PL pretraining and subword tokenization to achieve superior results compared to CodeBERT and VRepair. Without training from scratch, we tested the model, achieving a PP of 0.64%, as expected, compared to Fu et al.'s model results of 1%.

3.3. Improving VulRepair: ImpVulRepair

To improve VulRepair, we constructed ImpVulRepair by looking at the hyperparameters of VulRepair. The hyperparameters we focused on were the optimizer, learning rate, weight decay, batch size, encoder and decoder block sizes, and momentum. These hyperparameters influence the learning process of a model. We adjusted the hyperparameters to improve performance and searched for a current optimizer. When choosing an optimizer, various options were explored, including the LION optimizer, as discussed in research articles such as [32]. The decision was made based on the characteristics described for each optimizer, assessing their efficiency and determining their compatibility with the T5 neural network.

When selecting values for the other hyperparameters, we used a different approach. There are different methods for optimizing hyperparameters, such as manual search, random search, grid search, halving search, automated tuning, artificial neural network tuning, etc. [33]. We used a semi-random search along with information on optimization from a research paper by Chen et al. [32] to evaluate the hyperparameter values. We used the evaluation loss from training as our guide when selecting or tuning the hyperparameters since testing could take more than 5 h. We selected hyperparameter values based on the lowest loss. Training also takes time. Although it may not be ideal, we used only one epoch to find the best hyperparameter values. We used this method when reviewing the learning rate, weight decay, batch size, encoder/decoder block sizes, and betas. This was not an exhaustive grid search, and there could be other combinations that yield a better model.

3.3.1. Loss Function

A loss function is used in neural networks to evaluate the effectiveness of model training. During training, the model generates a predicted output, which is then compared with the target output. The target output represents the expected values based on the input provided. When the predicted output closely matches the target output, the loss is small. Therefore, an objective of neural networks during training is to minimize the loss.

3.3.2. Optimizer

Optimizers are algorithms that are used to minimize the loss function by changing the weights and learning rates of neural networks, thus reducing the loss between the true values and the predicted values [34]. Stochastic gradient descent, momentum, RMSprop, and Adam are several examples of optimizers. Google developed a PyTorch optimizer in 2023, as reported in [32]. It is supposedly better for transformer architectures. It is also memory-efficient. It seems like a viable option that we can use for ImpVulRepair. To obtain more information and a better understanding, we visited GitHub, which explained the optimizer and its implementation [35].

The Evolved Sign Momentum (LION) is an optimization algorithm for deep neural network training. It is a stochastic gradient descent method and utilizes the sign operator to

control updates [36]. It is memory-efficient compared to Adam. Currently, VulRepair uses AdamW, a popular optimizer that adds L2 regularization to the Adam optimizer. However, LION outperforms Adam, reduces training times, and is 2.3 times faster [32]. LION uses momentum and requires a smaller learning rate, larger batch sizes, and greater weight decay. It is simpler, faster, and has fewer hyperparameters than AdamW, with the optimal settings including a learning rate of 0.0001, weight decay of 0.01, momentum Beta1 set to 0.9, and momentum Beta2 set to 0.99 [34]. Therefore, we used the LION optimizer to determine whether it was a good fit for ImpVulRepair.

3.3.3. Learning Rate

The learning rate is a hyperparameter that determines the amount or step size for updating model parameters, such as weights and biases in neural networks, which influences the model's output. Larger learning rates lead to faster convergence toward optimal weights, while smaller rates result in slower convergence. Convergence means that the predicted values progressively approach the target values. If the learning rate is too high, it can cause instability during training, where the predicted and target values diverge instead of converge. If the learning rate is too small, it will take longer to reach the minimum loss. We tested several learning rates for our ImpVulRepair model. A lower learning rate is preferred for the LION optimizer. We tested learning rates of 2×10^{-6}, 1×10^{-5}, 2×10^{-5}, 2.25×10^{-5}, 2.75×10^{-5}, and 3×10^{-5}. The lowest learning rate that showed the lowest evaluation loss with the LION optimizer was 2×10^{-5}.

3.3.4. Weight Decay

Weight decay is a regularization technique that adds a penalty to the cost function to reduce overfitting. Overfitting occurs when the model learns too well or memorizes the data and is not able to generalize to or predict different data. The default weight decay for AdamW is 0.01, whereas the LION optimizer does not specify one and it is not set. The weight decay in LION is supposed to balance with the learning rate. Smaller learning weights and larger weight decays are preferred [32]. We tested weight decays of 0.01, 0.1, 0.001, 0.0001, and 0.00001. Since higher weight decays are preferable and there were negligible differences among weight decays of 0.001 and lower, we used a weight decay of 0.001, which resulted in low loss with a learning rate of 2×10^{-5}.

3.3.5. Batch Size

The batch size is the number of samples used in the network model before the weights in the model are updated. A batch represents a part of a data set that is used because the dataset may be too large to train at once. VulRepair starts with a batch size of 4. LION reports that it performs better with larger batch sizes [32]. We tested a batch size of 16. This was too large, as we received an out-of-memory message. So, we tested smaller sizes, including 1, 2, 3, 4, and 5. We selected 2 for the batch size, as it resulted in the lowest loss.

3.3.6. Encoder and Decoder Block Sizes

The T5 transformer is an encoder–decoder model, or a text-to-text transfer transformer, used in various natural language processing (NLP) tasks that converts text input into text output. Encoders are used to understand and extract relevant information from text input and then convert the input sequence into a vector. The decoder inputs this vector and predicts an output sequence. As noted by Fu et al., the T5 architecture is limited to 512 tokens, so anything larger is truncated. This means that the truncated portion is not processed by the machine learning model and its usefulness is lost. Similarly, VulRepair performs best if the patch to the vulnerability contains fewer than 20 tokens. Fu et al. stated that researchers should explore techniques that handle these larger token sizes for the vulnerability function and its repair [7]. We note that the model itself is provided in binary form, so we did not try to adjust the neural network layers.

However, according to the GitHub site of Fu et al., the encoder block size is set to 512, the maximum source length, and the decoder block size is set to 256, the maximum target length [22]. Since these values were provided, we adjusted them and used them for training and testing. We treated them as hyperparameters and observed their performance. We found that doubling the block size to 1024 for the encoder and 512 for the decoder produced stable results after adjusting other hyperparameters.

When we increased the size of the block, the training time increased. For example, it took 17 h to train for 57 epochs when we set the encoder to 1024 and the decoder to 512. However, it took 9 h to train the model for 75 epochs when the encoder was set at 512 and the decoder at 256. This is not surprising since padding can significantly slow training and should only extend to the longest example in a batch [37]. In addition, when we doubled the block sizes, the GPU and Google Drive disconnected at 57 epochs. In this case, we did not achieve stable results but proceeded with testing the model, achieving a perfect prediction of 40%. This is lower than the PP of Fu's VulRepair of 44%. We adjusted the hyperparameters to achieve stability in subsequent tests.

We also found that anything over 1536 for the encoder and 512 for the decoder block sizes immediately produced an out-of-error response. We tried different combinations, such as 1280/512 for the encoder/decoder, respectively, during training. Some combinations allowed the model to train but did not test well. Memory errors, GPU and file disconnections, and screen freezing could occur.

We also tried to eliminate out-of-memory errors by reducing the beam size. For example, we increased the encoder block size to 1280 and the decoder block size to 512 during training. We decreased the beams from 50 to 40. ImpVulRepair achieved a perfect prediction of 52.7%.

Although we could have further adjusted different combinations of hyperparameters to increase the perfect prediction percentage, we limited the training to 1024 for the encoder and 512 for the decoder. Although we tested a beam size of 40, we did not perform an exhaustive search to determine the optimal relationship between the encoder, decoder, and beam size. We could achieve stable testing by setting the beam size to 50 depending on other hyperparameter settings.

3.3.7. Momentum

Regarding optimizers, momentum refers to how the loss function accelerates convergence toward the minimum. We want a loss function to move horizontally, taking larger steps toward minimal loss while making smaller steps vertically. Momentum prevents gradient descent from oscillating as it moves toward the minimum. Betas are coefficients used in optimizers such as AdamW and LION. They are extensions of the momentum concept and improve efficiency and accuracy in training and testing transformer models [38]. There are two betas in both the LION and AdamW optimizers. They determine the update direction of the momentum. β-1 is the decay rate of the gradient, which is set to a default value of 0.9 in both the LION and AdamW optimizers. β-2 is the decay rate of the squared gradient, which is set to a default value of 0.99 in LION and 0.999 in AdamW [32].

We trained the ImpVulRepair model with the LION optimizer and its default beta values, achieving a PP of 55.7%. To yield better accuracy, we tuned LION's β-1 to 0.09, 0.6, 0.8, 0.9 (default for LION and AdamW), and 0.99, with 0.8 yielding the lowest loss. We tuned LION's β-2 to 0.9, 0.96, 0.98, 0.99 (default for LION), and 0.999 (default for AdamW), with 0.98 yielding the lowest loss. Thus, we used the optimal values, specifically 0.8 for β-1 and 0.98 for β-2, to train our model. Although we were able to reduce the training from 9 epochs to 8 epochs, our training was unstable. In our first attempt, the training log saved only one out of ten epochs. We defined instability as instances where data were not saved, the GPU and files disconnected, and the screen froze. In our second attempt, we received an OS error at the end of testing indicating that a file did not connect. However, ImpVulRepair achieved a PP of 51.93%. This was not our best result, and the testing was

unstable. Therefore, it is best to use the default beta values in LION, where β-1 is set to 0.9 and β-2 is set to 0.99, as they contribute to higher prediction percentages and stability.

4. Results

We conducted numerous training and tests that involved hundreds of hours. The longest training on a GPU took more than 17 h. Due to time and costs, we did not perform exhaustive tests for the parameters. Our best results took approximately 3 h to train and 6 h to test. We found a stable solution using Google Colab, Google Chrome, Google Drive, the LION optimizer developed by Google Brain, and by adjusting the hyperparameters, significantly improving VulRepair's perfect prediction from 44% to 56%.

4.1. LION Optimization

We conducted training and then testing and ensured that our model yielded stable results. The V100 GPU and Google Drive were not disconnected and the screen did not freeze. This took 2 h and 52 min to train and 5 h and 54 min to test. To see and compare the VulRepair and ImpVulRepair hyperparameters, refer to Table 2.

Table 2. A Comparison of hyperparameters used.

Hyperparameter Comparison		
Hyperparameter	**VulRepair**	**ImpVulRepair**
Optimizer	AdamW	LION
Learning Rate	2×10^{-5}	2×10^{-5}
Weight Decay	0.0	1×10^{-3}
Betas	(0.9, 0.999)	(0.9, 0.99)
Training Batch Size	4	2
Testing Batch Size	1	1
Epochs	75	10
Encoder Block Size	512	1024
Decoder Block Size	256	512
GPU	NVIDIA 3090	Colab V100
Perfect Prediction	**44%**	**56%**

4.2. Testing Deprecated AdamW

VulRepair uses AdamW as the optimizer. Using most of the hyperparameter settings from ImpVulRepair, we tested it by switching from the LION optimizer back to the AdamW optimizer. Because the original VulRepair setting used 75 epochs, we also increased the training from 10 to 75 epochs. In our setup, we obtained a 40% perfect prediction but the GPU and files disconnected. We stopped testing after 49 epochs. This setup performed worse than the Fu et al. model.

4.3. Testing PyTorch AdamW

Toward the end of testing VulRepair in Colab, we received messages in Colab indicating that AdamW was deprecated and should be replaced in the future with the PyTorch AdamW. According to Stack Overflow, the problem could be resolved by replacing "optimizer = AdamW..." with "optimizer = optim.AdamW..." [39]. The Hugging Face forum suggests using torch.optim.AdamW, which is the PyTorch implementation, or turning off the warning by setting no_deprecation_warning = True [40].

In the ImpVulRepair settings, we switched the LION optimizer to the PyTorch AdamW optimizer, that is, "torch.optim.AdamW," to avoid the deprecation warning. We also increased the epochs from 10 to 75 since the original VulRepair used 75 epochs. We

achieved a perfect prediction of 50.4%. According to the training log, the best results occurred after 14 epochs. We did not test after 55 epochs since the files and GPU eventually disconnected. Although the training was unstable, this setup performed better than the Fu et al. model with a PP of 44% but not better than the stable training of the LION optimizer with a PP of 56%.

5. Discussion

Using current libraries, a recently developed optimizer (LION) over AdamW, and hyperparameter adjustment, we created ImpVulRepair. It not only relies on updated libraries and software such as Python version 3.10.12 instead of version 3.9.7 but also uses Google Colab with a V100 rather than an NVIDIA 3090 as the graphics processing unit. Our ImpVulRepair can repair vulnerable functions with patches, achieving a perfect prediction of 56%, whereas VulRepair, developed in 2022, repairs vulnerable C/C++ functions with a PP of 44%.

ImpVulRepair includes the Google Chrome browser, Google Colab, Google Drive, the LION optimizer created by Google Brain, and the T5 architecture introduced by Google in 2019. There are many browsers we could have selected, but we leaned toward Google Chrome since we have used the Google Cloud Platform in the past, which recommends using Google Chrome. We used Google Colab as it provides access to GPUs, does not require software to install, and runs in a browser. In addition, Google Colab integrates with Google Drive, where we can easily save our work. We selected the LION optimizer after conducting research on optimizers and finding one that was recent as of 2023 and that works well with T5 models. Coincidentally, it was created by Google Brain. Our intention was not to show that Google products are in tune with each other but rather to try to resolve the issues of GPU and attached file disconnections. However, at the end of our testing, Google Colab started suggesting new GPUs that may perform better. They also now offer Tensor Processing Units (TPUs) version 2, developed by Google, which are suitable for neural networks handling large datasets and may outperform GPUs in some situations [28]. As of April 2024, the Colab TPU version 1 and the V100 GPU we used are being deprecated. ImpVulRepair could attain consistent results in areas where previous tests failed. Exploring and evaluating newly available Colab GPUs and TPUs might achieve improvements in patching vulnerabilities.

With regard to best practices, datasets should be prepared for training machine learning models. This includes cleaning the data by removing duplicates, that is, deduplication of data [41]. If duplicates are not removed, the model's efficiency may be inflated [42]. The datasets used for training and testing VulRepair contained duplicates. Although it may not be ideal, this should be documented and, at best, justified. We acknowledged it but did not justify it. According to [43], 60% of the pairs are duplicates. If the datasets are deduplicated, the performance decreases from 44.2% to 10.2%. Another study found that 40% of the data overlapped between the training set and the test set [44]. Their Mistral model used 7 billion parameters and 5 epochs, whereas VulRepair uses 220 million parameters and 75 epochs. Both use the AdamW optimizer. If overlap is used, a 57% performance prediction with a beam size of 5 could be achieved, but this PP reduces to 26% with deduplication. These studies indicate that our results would be lower with the deduplication of data.

Regarding performance, in one study, Yang et al. claimed that their TSBPORT, or type-sensitive patch BackPORTing, improved the success rate of VulRepair by 63.9% [45]. Learning models need sufficient data to train. An external threat to VulRepair is that it may not generalize well to other CWEs and datasets. Other datasets could be explored [7]. In one study, Nong et al. asserted that the lack of datasets for vulnerable programs prevents them from progressing. They used a new technique called VGX to generate "high-quality" vulnerability datasets that can improve VulRepair by 85% [46]. Because our ImpVulRepair is an extension of VulRepair, we note that these datasets could also be used to improve model training.

Like Fu et al., we did not conduct an exhaustive search to refine all hyperparameters, nor did we attempt to increase memory by adding extra equipment. In addition, further testing to improve vulnerability patching performance could be conducted in a cloud environment such as Google Cloud Platform (GCP), Amazon Web Services (AWS), or Microsoft Azure. Our goal was to show that software evolves and that updated algorithms, such as the use of a 2023 optimizer (LION), should be tested for increased model performance.

6. Limitations

We acknowledge several constraints of our study. The T5 model was neither developed nor trained by us. Adjustments to the deep learning layers, whether horizontally or vertically, were not feasible as we were limited to using only the binary file. We depended on the models provided by Fu et al. Additionally, Colab was chosen as it offers a conducive environment for enhancing VulRepair, with its ease of setup and pre-installed libraries. Our experiments required an Internet connection to Colab. Although an updated NVIDIA 4090 GPU is available, we did not access one on a local machine for testing. Despite achieving stable results, local testing could potentially eliminate Internet connectivity issues and enhance stability. Moreover, while some studies have shown improvements in VulRepair using AdamW, we did not explore the potential enhancements from the LION optimizer due to time and cost constraints. Such investigations may be more appropriate for researchers who manage their models.

7. Conclusions

Software vulnerabilities are defects in programming codes that can be exploited, necessitating tools like AIBugHunter to detect and repair such vulnerabilities. Developed by the ASWM research group, AIBugHunter scans C/C++ programs, identifies vulnerabilities by CWE type, estimates severity using CVSS scores, and suggests repairs. The tool integrates with Visual Studio Code and utilizes LineVul and VulRepair technologies for enhanced detection and repair capabilities. LineVul, a transformer-based model, predicts and locates vulnerabilities, whereas VulRepair, based on the T5 architecture, effectively repairs them, achieving a 44% perfect prediction rate. This research utilizes datasets like Big-Vul and CVEfixes to evaluate and improve the tool's performance. This study highlights the importance of pretraining with natural and programming languages, as well as the effectiveness of subword tokenization in improving repair predictions. Fu et al. significantly improved their model, illustrating superiority over competitors such as VRepair and CodeBERT. Their methodology was based on technologies from 2022. For example, they used Python 3.9.7, but version 3.10.12 is now available. They employed an NVIDIA GPU 3090, but the more recent 4090 model is now available, two years later.

After gaining insight into VulRepair through research, we obtained the GitHub model referenced above and modified its procedure. Our study used Google Colab, Google Drive, and a Google V100 GPU to develop and evaluate ImpVulRepair, rather than using a local machine for training and testing. We carefully fine-tuned and adjusted the hyperparameters surrounding the ImpVulRepair model for training and then integrated contemporary libraries. Unlike VulRepair, which uses the AdamW optimizer, our method adopts newer technologies, including the LION optimizer developed by Google Brain in 2023. This methodology allows us to better detect vulnerable functions and increases the perfect prediction percentage. Our model demonstrates a perfect prediction of 56% compared to VulRepair's 44%. This means that we can provide more patches for vulnerabilities and prevent more cyber attacks.

Author Contributions: All authors contributed equally to this project. Writing—original draft, B.K.; Writing—review and editing, Y.L. and J.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This material is based upon work supported by the National Science Foundation [Grant Number 2334243]. The funding source did not influence the study's preparation, design, data collection, analysis, or the decision to publish.

Institutional Review Board Statement: Not Applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: As referenced, ASWM research or Fu et al. provide their research for replication as accessed on 2 February 2024 at https://github.com/awsm-research/VulRepair.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

PL	Programming language
NL	Natural language
T5	Text-to-text transfer transformer
LION	Evolved Sign Momentum
CWE	Common Weakness Enumeration
CWSS	Common Vulnerability Scoring System

References

1. Fu, M.; Tantithamthavorn, C.; Le, T.; Kume, Y.; Nguyen, V.; Phung, D.; Grundy, J. AIBugHunter: A Practical tool for predicting, classifying and repairing software vulnerabilities. *Empir. Softw. Eng.* **2023**, *29*, 4. [CrossRef]
2. Mitre. CWE—About CWE, March 2024. Available online: https://cwe.mitre.org/about/index.html (accessed on 24 March 2024).
3. National Institute of Standards and Technology. National Vulnerability Database, NVD—Vulnerability Metrics, September 2022. Available online: https://nvd.nist.gov/vuln-metrics/cvss (accessed on 26 April 2024).
4. Fu, M.; Tantithamthavorn, C. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In Proceedings of the 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR), Pittsburg, PA, USA, 23–24 May 2022; pp. 608–620.
5. Li, Y.; Wang, S.; Nguyen, T.N. Vulnerability detection with fine-grained interpretations. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE, Athens, Greece, 23–28 August 2021; Association for Computing Machinery: New York, NY, USA, 2022; pp. 292–303.
6. What Is the T5-Model? | Data Basecamp, September 2023. Section: ML—Blog. Available online: https://databasecamp.de/en/ml-blog/t5-model (accessed on 26 March 2024).
7. Fu, M.; Tantithamthavorn, C.; Le, T.; Nguyen, V.; Phung, D. 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, Singapore, 14–18 November 2022; Association for Computing Machinery: New York, NY, USA, 2022; pp. 935–947.
8. Fan, J.; Li, Y.; Wang, S.; Nguyen, T.N. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In Proceedings of the 2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR), Seoul, Republic of Korea, 29–30 June 2020; pp. 508–512.
9. ZeoVan. ZeoVan/MSR_20_code_vulnerability_csv_dataset, April 2024. Original-Date: 2020-06-25T04:47:52Z. Available online: https://github.com/ZeoVan/MSR_20_Code_vulnerability_CSV_Dataset (accessed on 16 April 2024).
10. Bhandari, G.P.; Naseer, A.; Moonen, L. 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, Athens, Greece, 19–20 August 2021.
11. Yue, S. A Data-to-Product Multimodal Conceptual Framework to Achieve Automated Software Evolution for Context-rich Intelligent Applications. *arXiv* **2024**, arXiv:2404.04821. Available online: https://arxiv.org/abs/2404.04821 (accessed on 24 March 2024).
12. Seas, C.; Fitzpatrick, G.; Hamilton, J.A.; Carlisle, M.C. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In Proceedings of the 2024 IEEE 14th Annual Computing and Communication Workshop and Conference (CCWC), Las Vegas, NV, USA, 8–10 January 2024; pp. 484–490. [CrossRef]
13. Lin, G.; Wen, S.; Han, Qi.; Zhang, J.; Xiang, Y. Software Vulnerability Detection Using Deep Neural Networks: A Survey. *Proc. IEEE* **2020**, *108*, 1825–1848. [CrossRef]
14. Monperrus, M. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* **2018**, *51*, 1–24. [CrossRef]
15. Gupta, R.; Pal, S.; Kanade, A.; Shevade, S.K. DeepFix: Fixing Common C Language Errors by Deep Learning. In Proceedings of the AAAI Conference on Artificial Intelligence, San Francisco, CA, USA, 4–9 February 2017.

16. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*; Cohn, T., He, Y., Liu, Y., Eds.; Association for Computational Linguistics: New York, NY, USA, 2020; pp. 1536–1547.
17. Mashhadi, E.; Hemmati, H. Applying CodeBERT for Automated Program Repair of Java Simple Bugs. In Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), Madrid, Spain, 17–19 May 2021; pp. 505–509.
18. Marjanov, T.; Pashchenko, I.; Massacci, F. Machine Learning for Source Code Vulnerability Detection: What Works and What Isn't There Yet. *IEEE Secur. Priv.* **2022**, *20*, 60–76. [CrossRef]
19. Fu, M.; Tantithamthavorn, C.; Nguyen, V.; Le, T. ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We? In Proceedings of the 2023 30th Asia-Pacific Software Engineering Conference (APSEC), Seoul, Republic of Korea, 4–7 December 2023; IEEE Computer Society: Los Alamitos, CA, USA, 2023; pp. 632–636.
20. Alqaradaghi, M.; Kozsik, T. Comprehensive Evaluation of Static Analysis Tools for Their Performance in Finding Vulnerabilities in Java Code. *IEEE Access* **2024**, *12*, 55824–55842. [CrossRef]
21. Sutter, T.; Kehrer, T.; Rennhard, M.; Tellenbach, B.; Klein, J. Dynamic Security Analysis on Android: A Systematic Literature Review. *IEEE Access* **2024**, *12*, 57261–57287. [CrossRef]
22. Awsm-Research/VulRepair, March 2024. Original-Date: 2022-07-19T01:29:56Z. Available online: https://github.com/awsm-research/VulRepair (accessed on 26 March 2024).
23. Goodfellow, I.; Bengio, Y.; Courville, A. *Deep Learning*; MIT Press: Cambridge, MA, USA, 2016. Available online: http://www.deeplearningbook.org (accessed on 2 January 2024).
24. Kapoor, A.; Gulli, A.; Pal, S. *Deep Learning with TensorFlow and Keras*, 3rd ed.; Packt: Birmingham, UK, 2022.
25. Geron, A. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, 3rd ed.; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2023.
26. Shah, C. *A Hands-On Introduction to Machine Learning*; Cambridge University Press: Cambridge, UK, 2022.
27. Müller, A.C.; Guido, S. *Introduction to Machine Learning with Python: A Guide for Data Scientists*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2016.
28. Gadil, R.J. Maximizing Computing Power: A Guide to Google Colab Hardware Options, October 2023. Available online: https://medium.com/@romxzg/maximizing-computing-power-a-guide-to-google-colab-hardware-options-a68469415291 (accessed on 1 May 2024).
29. Fu, M. MickyMike/VulRepair Hugging Face, 2022. Available online: https://huggingface.co/MickyMike/VulRepair (accessed on 11 May 2024).
30. Wang, Y.; Wang, W.; Joty, S.; Hoi, S.C.H. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv* **2021**, arXiv:2109.00859. Available online: https://arxiv.org/abs/2109.00859 (accessed on 2 May 2024).
31. Zhou, X.; Kim, K.; Xu, B.; Han, D.; Lo, D. Large Language Model as Synthesizer: Fusing Diverse Inputs for Better Automatic Vulnerability Repair. *arXiv* **2024**, arXiv:2401.15459.
32. Chen, X.; Liang, C.; Huang, D.; Real, E.; Wang, K.; Liu, Y.; Pham, H.; Dong, X.; Luong, T.; Hsieh, Ch.; et al. Symbolic Discovery of Optimization Algorithms. *arXiv* **2023**, arXiv:2302.06675.
33. Pandian, S. A Comprehensive Guide on Hyperparameter Tuning and Its Techniques, February 2022. Available online: https://www.analyticsvidhya.com/blog/2022/02/a-comprehensive-guide-on-hyperparameter-tuning-and-its-techniques/ (accessed on 25 April 2024).
34. Bhaskar, Y. Lion Optimizer, November 2023. Available online: https://medium.com/@yash9439/lion-optimizer-73d3fd18abe9 (accessed on 30 March 2024).
35. Wang, P. Lucidrains/Lion-Pytorch, March 2024. Original-Date: 2023-02-15T04:24:19Z. Available online: https://github.com/lucidrains/lion-pytorch (accessed on 26 March 2024).
36. Keras. Keras LION Optimizers. 2024. Available online: https://keras.io/api/optimizers/lion/ (accessed on 24 March 2024).
37. Raffel, C.; Shazeer, N.; Roberts, A.; Lee, K.; Narang, S.; Matena, M.; Zhou, Y.; Li, W.; Liu, P.J. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *arXiv* **2023**, arXiv:1910.10683.
38. Wang, B.; Xia, H.; Nguyen, T.; Osher, S. How Does Momentum Benefit Deep Neural Networks Architecture Design? A Few Case Studies. *arXiv* **2021**, arXiv:2110.07034.
39. Cook, D. Implementation of AdamW Is Deprecated and Will Be Removed in a Future Version. Use the PyTorch Implementation torch.optim.AdamW, February 2023. Available online: https://stackoverflow.com/q/75535679 (accessed on 15 April 2024).
40. Panco. FutureWarning: This Implementation of AdamW Is Deprecated and Will Be Removed in a Future Version. Use the PyTorch Implementation torch.optim.AdamW Instead—Beginners, July 2023. Section: Beginners. Available online: https://discuss.huggingface.co/t/futurewarning-this-implementation-of-adamw-is-deprecated-and-will-be-removed-in-a-future-version-use-the-pytorch-implementation-torch-optim-adamw-instead/32283/3 (accessed on 15 Apr 2024).
41. Logan, S. Training AI in 2024: Steps & Best Practices, February 2024. Available online: https://www.twine.net/blog/training-ai/ (accessed on 24 April 2024).
42. Victoroff, S. Indico Data: Should We Remove Duplicates from a Data Set While Training a Machine Learning Algorithm (Shallow and/or Deep Methods)? February 2019. Available online: https://indicodata.ai/blog/should-we-remove-duplicates-ask-slater/ (accessed on 24 April 2024).

43. Zhou, X.; Kim, K.; Xu, B.; Han, D.; Lo, D. Out of Sight, Out of Mind: Better Automatic Vulnerability Repair by Broadening Input Ranges and Sources. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Lisbon, Portugal, 14–20 April 2024; ACM: New York, NY, USA, 2024; pp. 1–13.
44. De Fitero-Dominguez, D.; Garcia-Lopez, E.; Garcia-Cabot, A.; Martinez-Herraiz, J. Enhanced Automated Code Vulnerability Repair using Large Language Models. *arXiv* **2024**, arXiv:2401.03741.
45. Yang, S.; Xiao, Y.; Xu, Z.; Sun, C.; Ji, C.; Zhang, Y. Enhancing OSS Patch Backporting with Semantics. In Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, Copenhagen, Denmark, 26–30 November 2023; ACM: New York, NY, USA, 2023; pp. 2366–2380.
46. Nong, Y.; Fang, R.; Yi, G.; Zhao, K.; Luo, X.; Chen, F.; Cai, H. Vgx: Large-scale sample generation for boosting learning-based software vulnerability analyses. In Proceedings of the ICSE '24: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Lisbon, Portugal, 14–20 April 2024.