Boosting the Capability of Intelligent Vulnerability Detection by Training in a Human-Learning Manner

Shihan Dou†∥, Yueming Wu‡∥,* Wenzxuan Li‡, Feng Cheng§, Wei Yang‡, Yang Liu‡
†Fudan University, ‡Nanyang Technological University
§Huazhong University of Science and Technology, *University of Texas at Dallas

Abstract—Due to its powerful automatic feature extraction, deep learning (DL) has been widely used in source code vulnerability detection. However, although it performs well on artificial datasets, its performance is not satisfactory when detecting real-world vulnerabilities due to the high complexity of real-world samples. Meanwhile, almost all DL-based methods consider only designing a new model to detect vulnerabilities and ignore the importance of the training process. During their training phases, all training data are treated equally and fed into their models for training in a completely random order. However, the understanding ability of a model is different when learning different vulnerability knowledge. In other words, some vulnerability samples are difficult for the model to master while some samples are easy to learn.

In this paper, we propose to train DL-based vulnerability detection models in a human-learning manner, that is, start with the simplest samples and then gradually transition to difficult knowledge. Specifically, we design a novel framework (i.e., Humer) that can enhance the detection ability of DL-based vulnerability detectors. To validate the effectiveness of Humer, we select five state-of-the-art DL-based vulnerability detection models (i.e., TokenCNN, VulDeePecker, StatementGRU,ASTGRU, and Devign) to complete our evaluations. Through the results, we find that the use of Humer can increase the F1 of these models by an average of 10.5%. Moreover, Humer can make the model detect up to 16.7% more real-world vulnerabilities. Meanwhile, we also conduct a case study to uncover vulnerabilities from real-world open source products by using these enhanced DL-based vulnerability detectors. Through the results, we finally discover 281 unreported vulnerabilities in NVD, of which 98 have been “silently” patched by vendors in the latest version of corresponding products, but 159 still exist in the products.

Index Terms—Vulnerability Detection, Deep Learning, Human Learning

I. INTRODUCTION

Recently, as the complexity of computer software systems has increased, the number of potential security vulnerabilities has shown an increasing trend. In 2021, National Vulnerability Database (NVD) reported that, starting from 2018, the number of disclosed vulnerabilities has exceeded 15,000 for four consecutive years [1]. Although some of these vulnerabilities have been fixed, security incidents caused by vulnerabilities are still emerging in endlessly. For example, in May 2017, “WannaCry” ransomware [2] exploited a dangerous vulnerability (i.e., “EternalBlue”) disclosed by National Security Agency (NSA) in March of the same year to attack Windows operating system and invade users’ hosts. The ransomware has spread to at least 150 countries and regions, causing hundreds of millions of losses to governments, enterprises, universities, and other industries. To resist these security threats, source code vulnerability detection has become the most effective solution.

Generally speaking, source code vulnerability detection methods can be classified into two main categories (i.e., the code-similarity-based methods [3, 4, 5, 6, 7] and the pattern-based methods [8, 9, 10]. Methods in the first category focus on detecting vulnerabilities caused by code cloning. They leverage different similarity calculation techniques to measure the similarity between known vulnerabilities and the source code to be detected. If the similarity is higher than a preset threshold (e.g., 80%), the source code will be reported as a vulnerability. In other words, code-similarity-based methods can only detect known vulnerabilities.

To detect more vulnerabilities, pattern-based methods are designed. Among these methods, deep learning (DL)-based approaches [11, 12, 13, 14, 15, 16, 17, 18] have attracted wide attention since they do not require experts to manually define features and can automatically generate vulnerability patterns from program source code. For example, VulDeePecker [11] treats the program slices of source code as text and applies a bidirectional long short-term memory (BLSTM) neural network to detect vulnerabilities. Funded [18] first conducts complex program analysis to extract augmented abstract syntax tree (AST) of source code, and then uses a graph neural network (GNN) to train a vulnerability detector. According to their papers’ results, they can achieve ideal performance on detecting vulnerabilities in Software Assurance Reference Dataset (SARD) [19]. But in fact, vulnerabilities in SARD are synthetic and are only used for simple academic research. To check the feasibility of DL-based methods on real-world vulnerability detection, a recent study [20] uses several real-world vulnerability datasets and conducts a comparative evaluation on certain DL-based vulnerability detectors. Through the results, they find that due to the high complexity of real-world vulnerabilities, the detection performance of evaluated detectors can drop a lot.

Meanwhile, we also observe that almost all DL-based vulnerability detectors only focus on designing a new model and neglect the importance of their training phases. They train models in a straightforward manner, that is, all training samples are treated equally and presented in a completely random order during training. However, a model has different
understanding capabilities when learning different vulnerability knowledge. It can converge quickly when learning a certain type of vulnerability knowledge, but it may converge slowly when targeting some other types of vulnerability knowledge (Section II). In other words, some vulnerability samples are difficult for the model to understand and some samples are easy to learn. For humans, there is a similar phenomenon in our learning process, that is, our ability to understand different knowledge is also different. However, in our human learning process, we do not directly learn difficult knowledge at the beginning, but start from understanding simple knowledge, and then transition to the difficult content. That is to say, our human learning method is phased, unlike DL-based vulnerability detection, where samples are completely randomly inputted into the model for training. In fact, previous studies [21], [22], [23], [24] have shown that if a machine learning model can be trained in a phased learning way, its learning effect will be better. Researchers refer to this type of learning method as Curriculum Learning, which is first proposed by Bengio et al. [25]. Its original formulation can be viewed as a continuation method, which can help to find better local minima of a highly non-convex criterion and can reach faster convergence to a minimum of the training criterion [25].

In this paper, we draw inspiration from Curriculum Learning and propose to model human learning procedures to train DL-based vulnerability detectors. Specifically, we not only simulate human phased learning, but also consider the review and consolidation in human learning process to train DL-based vulnerability detectors. For knowledge phased learning, we introduce two difficulty computation techniques of vulnerability samples. The first is to start from the model itself and focus on the model’s ability to recognize vulnerability samples. The second is to start from the vulnerability sample and focus on the code complexity of the sample itself. After sorting all samples by difficulty, we divide them into different buckets according to the degree of difficulty, and then start training from the simplest bucket. For knowledge review, before learning the next bucket, we put the samples of the previous bucket into this bucket first, and then learn with this bucket to achieve the purpose of review. For knowledge consolidation, we record all mispredicted samples in the training process of each bucket, and apply semantic-equivalent code transformations to generate multiple variants of these samples. These variant samples constitute an error book and are used to fine-tune the model to strengthen the learning of ununderstood knowledge.

We implement a novel framework namely Humer and select five state-of-the-art DL-based vulnerability detectors (i.e., TokenCNN, VulDeePecker, StatementGRU, ASTGRU, and Devign) to evaluate Humer. Through the results, we find that using Humer to train these models can make them detect 12.6% more vulnerabilities on average. Meanwhile, the F1 values also increase by an average of 10.5% after using Humer. To further demonstrate the effectiveness of Humer, we conduct a case study on seven popular open source products with a total number of lines of code exceeding 58 million. Specifically, we first train five models (i.e., TokenCNN, VulDeePecker, StatementGRU, ASTGRU, and Devign) with Humer to obtain enhanced detectors, and then scan vulnerabilities on downloaded products. Through the scanning reports, we finally detect 281 vulnerabilities that are not reported in NVD. Among them, 98 have been “silently” patched by vendors in the latest version of corresponding products, 24 vulnerabilities have been deleted, and the other 159 still exist in the products. As for improvement, the scanning results indicate that using Humer can make them (i.e., TokenCNN, VulDeePecker, StatementGRU, ASTGRU, and Devign) detect 37 more unreported vulnerabilities on average.

In summary, this paper makes the following contributions:

- As far as we know, we are the first to model human learning procedures into the training of DL-based vulnerability detection.
- We implement Humer, a model-agnostic framework that can enhance the effectiveness of DL-based vulnerability detection models.
- We select five state-of-the-art DL-based vulnerability detectors (i.e., TokenCNN, VulDeePecker, StatementGRU, ASTGRU, and Devign) and conduct evaluations using a real-world vulnerability dataset. Experimental results report that the use of Humer can make them detect 12.6% more vulnerabilities on average.
- We conduct a case study on more than 58 million lines of code to further demonstrate the practicability of Humer. Through the scanning results, we discover 281 vulnerabilities that are not reported in NVD.

II. Motivation

Due to the powerful capability of automatic feature extraction, deep learning (DL) has been widely used in vulnerability detection. However, in these DL-based vulnerability detectors, all training samples are randomly presented to the model, ignoring the complexity of the vulnerability samples and the current learning state of the model. In fact, we humans do not learn like this, but start learning from simple knowledge, and then move on to difficult knowledge. The main reason why we humans learn this way is because our ability to understand different knowledge is different. Some knowledge can be easily mastered, while some knowledge we need to spend more time to understand. For those difficult knowledge, we should first learn their basic principles (simple knowledge). After mastering these principles, we can understand the difficult knowledge faster and more thoroughly. In order to check whether there are similar phenomena in DL-based vulnerability detection models, we perform a simple investigation in this section. Specifically, we mainly answer the following questions:

Is the ability of DL-based vulnerability detection models to learn different vulnerability knowledge different?

To answer the raised question, we use the vulnerability data in a prior study [26] as the test object. The authors collect a total of 74 different types of vulnerable functions from SARD, and we select the vulnerability types with more than
3,100 vulnerabilities. There are five types of vulnerabilities that meet our criteria (i.e., CWE-121, CWE-122, CWE-134, CWE-190, and CWE-191). The descriptions of these five types of vulnerability are described in Table I. To commence our research, we randomly choose 3,100 vulnerability samples from each type of vulnerability data, and randomly select 1,100 samples from normal data to form our experimental dataset. For each type of vulnerability, we randomly select 100 normal functions and 100 vulnerabilities as the validation set, and other samples (1,000 normal functions and 3,000 vulnerabilities) as the training set. We mainly conduct three experiments. The first is to train a model by using 1,000 normal functions and 1,000 vulnerabilities and then record the loss during validation phase. The second is to add 1,000 vulnerabilities to the training set and then use these samples to train the model and record the loss during validation phase. The third is to continue to add 1,000 vulnerabilities to the training set and then repeat the previous process. For DL-based vulnerability detection model, we choose TokenCNN [17] since it has been widely used as a comparative model by many other researchers [13, 20]. It first applies lexical analysis to parse the source code of a program into tokens and then inputs them into a convolutional neural network to train a vulnerability detector.

The experimental results are shown in Figure 1. It can be seen that although the size of the training set is the same, the model has different learning capabilities for different types of vulnerability (i.e., the loss is different). When the vulnerability data is added, the loss generally becomes lower, indicating that the model learns more vulnerability knowledge. But for some types of vulnerabilities, even if the number of vulnerabilities increases to 2,000, the model’s ability is still not good enough. For example, for CWE-191, when the training set consists of 2,000 vulnerabilities, the loss in the first epoch is 0.4583. For CWE-190, even if there are only 1,000 vulnerabilities in the training set, its loss in the first epoch is lower than 0.4583, only 0.4332. Such results indicate that TokenCNN has different understanding capabilities when learning different types of vulnerability knowledge. Compared to CWE-191, TokenCNN has a stronger understanding ability on learning knowledge in CWE-190. Based on the observation, we propose to train DL-based vulnerability detection models in a meaningful order rather than a random order.

### III. Design of Humer

In this section, we introduce Humer, a novel framework that models human learning procedures to enhance the effectiveness of DL-based vulnerability detection.

#### A. Overview

As shown in Figure 2, Humer consists of three main components: Difficulty Calculator, Curriculum Learner, and Data Augmenter.

- **Difficulty Calculator**: Given a training dataset, we first compute the difficulty of each sample and divide them into several buckets according to the degree of difficulty.
- **Curriculum Learner**: Given sorted buckets, we start training from the simplest bucket, and then learn the more

| CWE   | Description                                                                 |
|-------|-----------------------------------------------------------------------------|
| CWE-121 | A stack-based buffer overflow condition is a condition where the buffer being overwritten is allocated on the stack. |
| CWE-122 | A heap overflow condition is a buffer overflow, where the buffer that can be overwritten is allocated in the heap portion of memory. |
| CWE-134 | The software uses a function that accepts a format string as an argument, but the format string originates from an external source. |
| CWE-190 | The software performs a calculation that can produce an integer overflow or wraparound, when the logic assumes that the resulting value will always be larger than the original value. |
| CWE-191 | The product subtracts one value from another, such that the result is less than the minimum allowable integer value, which produces a value that is not equal to the correct result. |
difficult buckets in turn. For each bucket, we record all mispredicted data during training and use Data Augmentor to generate the corresponding variants, and then fine-tune the model using these variants.

- **Data Augmentor**: Given mispredicted data, we apply semantic-equivalent code transformations to construct the corresponding variant data.

B. Difficulty Calculator

Because a model has different abilities on understanding different vulnerability knowledge, we compute a sample’s difficulty by analyzing the model’s ability on recognizing it. However, the runtime overhead of analyzing a model is not lightweight. Therefore, we design another alternative model-independent difficulty calculation method from the perspective of we humans. That is, we compute the difficulty of a sample by analyzing the source code complexity directly.

1) Model-based Difficulty Calculator

In this part, we pay attention to the model’s ability on recognizing vulnerability samples. In other words, a sample with a stronger recognition ability will be considered easy to learn, while a sample that is not easily recognized will be considered a difficult sample.

To complete difficulty calculation, we first split our training set $D$ into $M$ subsets uniformly as $\{d_i : i = 1, 2, ..., M\}$, and then train $M$ corresponding submodels $\{m_i : i = 1, 2, ..., M\}$. Note that each submodel only sees $1/M$ samples in the training set. After obtaining $M$ submodels, we input samples in $D$ to these submodels to obtain the corresponding predicted probabilities. The model considers a sample as a vulnerability only when the predicted probability is greater than 0.5. Therefore, we take the part with the predicted probability greater than 0.5 as the submodel’s confidence of identifying the vulnerability sample. If it is a vulnerability sample, the greater the predicted probability, the greater the confidence that the submodel classifies the sample as a vulnerability. On the contrary, if the sample is normal, the lower the predicted probability, the greater the confidence that the submodel distinguishes the sample as normal.

In addition, for each submodel, its prediction correct rate generally does not reach 100%, that is, some samples may be predicted incorrectly. For example, assuming that subset $d_1$ consists of 1,000 vulnerability samples and 1,000 normal samples. After completing the training of submodel $m_1$ by using these samples, we then input them to submodel $m_1$ for testing. In other words, these 2,000 samples are both training set and testing set. For the testing results, suppose $m_1$ predicts that 1,100 samples are vulnerabilities, but in fact, only 800 samples are vulnerabilities, and the other 300 samples are normal samples. It means that the correct rate of submodel $m_1$ in predicting vulnerabilities is $800/1100 \approx 72.7\%$, and there...
is a 27.3% probability that the prediction is wrong.

To obtain more credible computation results, we consider both the prediction correct rate of the submodel and the discrimination confidence of the sample. That is, the higher the prediction correct rate of the submodel itself and the greater the confidence in distinguishing the sample, the stronger the model’s ability to recognize the sample. The calculation formula is as follows:

\[
DS_{s \in d_j} = \begin{cases} 
- \sum_{i \in j} \frac{TP_i}{TP_i + FP_i} \times (p_i - 0.5), & \text{label}(s) = 1 \\
- \sum_{i \notin j} \frac{TN_i}{TN_i + FN_i} \times (0.5 - p_i), & \text{label}(s) = 0
\end{cases}
\]

For each subset \(d_j\), we first use it to train a submodel \(m_i\). After training, we then input it to \(m_i\) for testing and record the true positives (i.e., TP, #samples correctly predicted as vulnerable), true negatives (i.e., TN, #samples correctly predicted as normal), false positives (i.e., FP, #samples incorrectly predicted as vulnerable), and false negatives (i.e., FN, #samples incorrectly predicted as normal). By this, we can obtain the prediction correct rate of all submodels. More specifically, for each submodel, we compute the correct rate in predicting vulnerabilities by analyzing the quotient of TP and (TP+FP). For normal samples, we compute the quotient of TN and (TN+FN) as the prediction correct rate. As shown in Figure 1, given a sample \(s\), suppose it belongs to subset \(d_j\) and its label is 1 (i.e., \(s\) is a vulnerability). To compute its difficulty, we input it to all submodels except submodel \(m_j\) because \(m_j\) has already seen it during training. For each submodel, after obtaining the corresponding predicted probability \(p\) of \(s\), we can calculate the discrimination confidence by computing \(p\) minus 0.5 since \(s\) is a vulnerability sample. After finishing computing the prediction correct rate of the submodel and the discrimination confidence of the sample, the submodel’s ability on recognizing the sample will be computed as their product. Each submodel will calculate a recognition ability score, and the negative of the sum of these scores (i.e., \(N\)-1 scores) is the difficulty score of the sample. The lower the difficulty score, the stronger the model’s ability to distinguish the sample, making it easier to learn.

2) Code-based Difficulty Calculator

In this part, we compute the difficulty of a sample by analyzing the code complexity from the source code directly. Samples with lower code complexity are regarded as simple samples, and samples with higher complexity are treated as difficult samples.

According to prior work [28, 29], the maintainability of a program can give an overall picture of program complexity. The lower complexity of the program, the easier it is to maintain. To complete our difficulty calculation, we adopt Maintainability Index as the standard to evaluate the code complexity of samples. Maintainability Index [30, 31, 32] is more of an empirical measure, it has been developed over the years with consultants from Hewlett-Packard and its software teams [33]. This index weighs Cyclomatic Complexity and Halstead Volume [35] against the number of Source Lines of Code in the program. These three metrics are described as follows:

- **Source Lines of Code**: It is the most straightforward metric used to measure the size of the program. It is obtained by counting the total number of lines in the source code excluding comments and blank lines. The more lines, the more complex the code.
- **Cyclomatic Complexity**: It measures how much control flow exists in a program. It is a count of the number of decisions in the source code. The higher the count, the more complex the code.
- **Halstead Volume**: It measures how much information is in the source code. It looks at how many variables are used and how often they are used in the source code. The larger the volume, the more complex the code.

According to prior research [30, 31, 32], if we define the Source Lines of Code as \(L\), the Cyclomatic Complexity as \(G\), and the Halstead Volume as \(V\), then the Maintainability Index (MI) can be calculated as follows:

\[
MI = 171 - 5.2 \ln V - 0.23G - 16.2 \ln L
\]

The more complex the code, the lower the Maintainability Index, and the greater the difficulty score.

C. Curriculum Learner

After evaluating the difficulty of all samples in the training set, we sort them by their difficulty scores and divide them into \(N\) buckets uniformly as \(\{b_i : i = 1, 2, ..., N\}\). By this, samples in the training set are collected into \(N\) different levels of difficulty, ranging from \(b_1\) (the easiest) to \(b_N\) (the hardest).

After completing the bucket assignment, we start our training from the easiest bucket. When the training reaches convergence or a preset number of a maximum epoch, we stop it and transfer to the next training stage. Specifically, the preset maximum number of epochs in our experiments is 10, and the training will be treated as converged if the training loss stops to decrease. Before starting the next training, we first add the samples of the previous bucket to the next bucket and apply transfer learning to learn this bucket, so as to realize the knowledge transfer between the two buckets. Since we allocate \(N\) buckets, we perform a total of \(N\)-1 transfer learning.

In our human learning process, it is difficult to understand all knowledge at once. For the knowledge that we don’t understand, we humans generally need to learn many times to master it. We model this learning procedure into the training of DL-based vulnerability detection. Specifically, for each bucket, we record the mispredicted samples during training, and leverage our Data Augmenter module to generate the corresponding semantic-equivalent variant samples. These variant samples constitute an error book and are used to fine-tune the model to strengthen the learning of ununderstood knowledge.
output of Curriculum Learner is a model that can be used to detect vulnerability.

D. Data Augmenter

After collecting mispredicted samples, we feed them into our Data Augmenter to output some semantic-equivalent variant samples. The implementation of Data Augmenter is based on several designed rules. By using these rules, we can mutate the source code of mispredicted samples to generate new semantically equivalent samples.

![Diagram of code transformation rules]

Fig. 4: Code transformation rules used in Data Augmenter

On the one hand, to make a model resilient to common code modifications while preserving program semantics, some DL-based vulnerability detection techniques apply abstraction and normalization to the source code. For example, they may map user-defined variables to symbolic names in a one-to-one manner (e.g., VAR). On the other hand, to make a model more effective on vulnerability detection, researchers may consider distilling the program semantics into different code representations such as abstract syntactic tree (AST) and control flow graph (CFG). These code representations will be fed into different neural networks such as graph neural network (GNN) to train vulnerability detectors. Therefore, we do not select those simple code transformations such as variable rename but design several rules that can generate new semantic-equivalent variants with different control flow structures. Our rule design is based on the observation that the same algorithm can have syntactically different implementations. More specifically, we implement a total of five code transformation rules as shown in Figure 4:

- **Rule1 (Introduce new else branch):** This rule introduces new else branching statement to an existing if conditional statement or reverse the condition in the if conditional statement.
- **Rule2 (Conditional expression with conjunction):** A conjunct if statement with conditions C1 and C2 can be rewritten as a nested if structure, which contains conditions C1 and C2 individually.
- **Rule3 (Introduce guard around for loop):** A program containing a for loop statement can be changed to a new program structure by introducing guard around the loop.
- **Rule4 (Introduce guard around while loop):** Similar to Rule3, a program containing a while loop statement can be mutated into a new program structure by introducing guard around the loop.
- **Rule5 (Conditional break inside while):** A program that loops until a condition C1 is satisfied can be transformed into another infinite loop program, with a C1 conditional break instruction inside the loop’s body.

Given a program, we first traverse it to search for the statements that satisfy the code transformation rules. After collecting all the code statements that can be mutated, we perform code transformation on these codes according to the conforming rules. To implement code transformation with different levels of difficulty, we design two types of transformation combinations. The first type is simple transformation, that is, transform only one code segment in the program that can be mutated. The second type is hard transformation, that is, all codes that satisfy the transformation are transformed.

To better illustrate our code transformation phase, we take CVE-2018-7751 as an example and show the detailed procedure in Figure 3. After analyzing the program, we find that there are four code fragments in CVE-2018-7751 that meet the transformation conditions. Specifically, lines 5 to 6, lines 9 to 10, and lines 11 to 12 all meet Rule1, and lines 7 to 13 meet Rule4 and Rule5. To obtain variant samples, we first use Rule1 to apply code transformation on lines 5 to 6, and the other codes remain unchanged. In this way, a variant can be generated. Repeating the above processes, we can construct five variants, which are called simple transformations. To
generate more difficult transformation codes, we transform all the codes that satisfy the transformation rules, which are called hard transformations. In one word, after inputting CVE-2018-7751 into Data Augmenter, we can obtain seven semantic-equivalent variant samples with different control flow structures.

IV. EXPERIMENTS

Since Humer is composed of three main modules (i.e., Difficulty Calculator, Curriculum Learner, and Data Augmenter), we conduct experiments to evaluate Humer from these three perspectives. For Difficulty Calculator, we focus on the comparative experimental analysis of model-based and code-based difficulty calculators. For Curriculum Learner, we focus on the comparative experimental analysis of learning without curriculum and learning with curriculum. For Data Augmenter, we focus on the comparative experimental analysis of learning without data augmentation and learning with data augmentation.

TABLE II: Descriptions of used dataset in our experiments

|      | #Vulnerable Functions | #Normal Functions |
|------|-----------------------|-------------------|
| Qemu | 7,479                 | 1,0070            |
| Ffmpeg| 4,981               | 4,788             |
| Total | 12,460               | 14,858            |

A. Experimental Setup

1) Dataset
To test Humer on real-world vulnerability detection, we select a widely used open source vulnerability dataset which consists of 12,460 vulnerabilities and 14,858 normal functions. A team of four professional security researchers spent 600 man-hours to construct it. Functions in the dataset are mainly obtained from two open source software, namely qemu and ffmpeg. A total of 7,479 vulnerable functions and 10,070 and normal functions are analyzed from qemu, and a total of 4,981 vulnerable functions and 4,788 normal functions are obtained from ffmpeg. The average number of lines of code (LOC) of these vulnerabilities and normal functions are 63 and 57, respectively.

2) Selected DL-based Vulnerability Detectors
Since a program can be characterized by different code representations such as abstract syntactic tree (AST) and control flow graph (CFG), we select five state-of-the-art DL-based vulnerability detection methods corresponding to five different code representations, respectively. By this, we can achieve a more comprehensive evaluation of Humer. These five detector\footnote{For convenience, we name the unnamed models based on their code representations (e.g., token) and neural networks (e.g., CNN).} are as follows:

- **TokenCNN** \cite{17}: It is a token-based vulnerability detector. After obtaining the tokens of a program by lexical analysis, TokenCNN inputs them into a convolutional neural network (CNN) to train a model for vulnerability detection.
- **VulDeePecker** \cite{11}: It is a slice-based vulnerability detector. After extracting the slices of a program by static analysis, VulDeePecker inputs them into a bidirectional long short-term memory (BLSTM) network to train a model for vulnerability detection.
TABLE III: Experimental results of Humer with model-based and code-based Difficulty Calculators. DC denotes Difficulty Calculator, CL denotes Curriculum Learner. To complete model-based difficulty calculation, we first split the training set into $M$ subsets and train $M$ submodels. To complete code-based difficulty calculation, we directly analyze the code complexity of all samples in the training set. After obtaining all samples’ difficulty, we sort them and divide them into $N$ buckets for Curriculum Learner.

| TokenCNN | Baseline | Humer (model-based DC) | Humer (code-based DC) |
|----------|----------|------------------------|-----------------------|
| M for DC | None     | 3 5 10                 | 3 5 10               |
| N for CL | None     | 3 5 10                 | 3 5 10               |
| Accuracy | 0.458    | 0.510 0.540 0.539      | 0.500 0.519 0.525    |
| Recall   | 0.510    | 0.673 0.677 0.662      | 0.665 0.642 0.620    |
| Precision| 0.416    | 0.448 0.497 0.477      | 0.448 0.469 0.469    |
| F1       | 0.458    | 0.538 0.573 0.554      | 0.535 0.542 0.534    |

| VulDeePecker | Baseline | Humer (model-based DC) | Humer (code-based DC) |
|--------------|----------|------------------------|-----------------------|
| Accuracy     | 0.532    | 0.635 0.554 0.580      | 0.582 0.547 0.587    |
| Recall       | 0.577    | 0.678 0.658 0.716      | 0.672 0.743 0.606    |
| Precision    | 0.472    | 0.585 0.501 0.524      | 0.525 0.497 0.542    |
| F1           | 0.519    | 0.628 0.569 0.605      | 0.590 0.595 0.572    |

| StatementGRU | Baseline | Humer (model-based DC) | Humer (code-based DC) |
|--------------|----------|------------------------|-----------------------|
| Accuracy     | 0.537    | 0.649 0.572 0.600      | 0.595 0.565 0.600    |
| Recall       | 0.592    | 0.722 0.677 0.730      | 0.694 0.765 0.627    |
| Precision    | 0.488    | 0.572 0.518 0.539      | 0.538 0.507 0.552    |
| F1           | 0.535    | 0.638 0.587 0.620      | 0.606 0.610 0.587    |

| ASTGRU       | Baseline | Humer (model-based DC) | Humer (code-based DC) |
|--------------|----------|------------------------|-----------------------|
| Accuracy     | 0.580    | 0.627 0.617 0.650      | 0.621 0.648 0.671    |
| Recall       | 0.547    | 0.685 0.680 0.624      | 0.691 0.629 0.649    |
| Precision    | 0.540    | 0.567 0.564 0.622      | 0.572 0.616 0.642    |
| F1           | 0.543    | 0.621 0.617 0.623      | 0.626 0.622 0.646    |

| Devign       | Baseline | Humer (model-based DC) | Humer (code-based DC) |
|--------------|----------|------------------------|-----------------------|
| Accuracy     | 0.561    | 0.660 0.669 0.659      | 0.664 0.664 0.645    |
| Recall       | 0.572    | 0.678 0.703 0.682      | 0.681 0.669 0.688    |
| Precision    | 0.534    | 0.602 0.603 0.599      | 0.583 0.593 0.574    |
| F1           | 0.553    | 0.638 0.649 0.638      | 0.638 0.626 0.626    |

- **StatementGRU** [26]: It is a statement-based vulnerability detector. *StatementGRU* treats the statements of a program as sentences and directly inputs them into a bidirectional gated recurrent unit (BGRU) network to train a model for vulnerability detection.
- **ASTGRU** [27]: It is a tree-based vulnerability detector. After obtaining the AST of a program by static analysis, *ASTGRU* first applies preorder traversal search algorithm to transform the AST into a sequence and then inputs the sequence into a bidirectional gated recurrent unit (BGRU) network to train a model for vulnerability detection.
- **Devign** [13]: It is a graph-based vulnerability detector. After extracting the abstract syntactic tree (AST), control flow graph (CFG), data flow graph (DFG), and natural code sequence (NCS) of a program, *Devign* inputs them into a graph neural networks (GNN) to train a model for vulnerability detection.

3) Implementations

We run all experiments on a server with 32 cores of CPU and a GTX 2080Ti GPU. For the dataset, we first randomly divide them into ten subsets, then the eight subsets are used for training, the other one subset is used for validating, and the last subset is used for testing.

4) Metrics

We adopt the widely used metrics to evaluate the detection results of *Humer* as follows:
- **True Positive** (TP): the number of samples correctly predicted as vulnerable
- **True Negative** (TN): the number of samples correctly predicted as normal
- **False Positive** (FP): the number of samples incorrectly classified as vulnerable
- **False Negative** (FN): the number of samples incorrectly classified as normal
- **Accuracy**=(TP+TN)/(TP+TN+FP+FN)
- **Recall**=TP/(TP+FN)
- **Precision**=TP/(TP+FP)

F1 = 2*(Precision*Recall)/(Precision+Recall)

B. For Difficulty Calculator

As aforementioned, we not only design a Difficulty Calculator from the perspective of the model itself, but also implement another alternative Difficulty Calculator by directly analyzing the code complexity of vulnerability samples. Therefore, in this part, we focus on evaluating the detection effectiveness of *Humer* with these two Difficulty Calculators. For model-based Difficulty Calculator, we first split the training set into $M$ subsets and train $M$ submodels. These submodels are used to evaluate the difficulty of samples in the training set.
In our experiments, we select three different $M$ (i.e., 3, 5, and 10) to commence model-based difficulty calculation. For code-based Difficulty Calculator, we do not split the training set but directly analyze the code complexity of all samples. After computing the difficulty, we sort these samples according to their difficulty scores and divide them into $N$ buckets uniformly. In this way, we can train the model in a meaningful order, that is, learn from the simplest bucket and then transition to difficult buckets. We also chose three different $N$ (i.e., 3, 5, and 10) to train the model.

Table III presents the detailed results of Humer with model-based and code-based Difficulty Calculators. Through these results, we obtain several findings. The first is that when Humer adopts model-based Difficulty Calculator, the detection effectiveness is more improved. For example, when training VulDeePecker, if Humer uses model-based calculator to compute the difficulty, the F1 can be increased to 62.8%, but it can only be increased to 56.5% when adopts code-based calculator. It is reasonable because we humans think that complex and difficult samples are not necessarily difficult for the model. In fact, our training object is a model, not humans. If the difficulty is calculated from the perspective of the model itself, the difficulty will be more suitable for the model, making the training of the model more meaningful.

The second is that when different $M$ and $N$ are selected, the improved effectiveness is also different. This is normal, because when different $M$ is used (i.e., the training set is divided into different numbers of subsets), the difficulty of the calculated sample may be different due to the different number of submodels. This can affect the subsequent difficult sorting, and then affect the final training effectiveness. When using different $N$ (i.e., the sorted samples are divided into different numbers of buckets), the number of samples in each bucket will be different, and different numbers of training samples will make the final training effectiveness different. In fact, we also observe that the overall effectiveness is generally best when it is divided into three subsets (i.e., $M = 3$) to calculate the difficulty. When the number of subsets increases, the number of vulnerabilities in each subset will decrease. If the number of vulnerabilities is small, after training the submodel, the submodel itself will not learn much knowledge of vulnerabilities, which may lead to errors when using these submodels to evaluate the difficulty. We will choose more parameters in the future to find the most suitable combination.

C. For Curriculum Learner

In this part, we focus on the comparative analysis of DL-based vulnerability detectors with and without Humer. A shown in Table III since we select three different $M$ and three different $N$ to test Humer, we can obtain a total of twelve experimental results (i.e., nine experimental results for model-based Difficulty Calculator and three experimental results for code-based Difficulty Calculator) for each model. To present it more clearly, we only show the lowest and highest results among these twelve results in Figure 6.

Table IV: The improvement of DL-based vulnerability detectors (i.e., TokenCNN, VulDeePecker, StatementGRU, ASTGRU, and Devign) after using Humer to train them.

|        | Accuracy | Recall | Precision | F1    |
|--------|----------|--------|-----------|-------|
| TokenCNN | 8.2%     | 16.7%  | 8.1%      | 11.5% |
| VulDeePecker | 10.3%   | 10.1%  | 11.3%     | 10.9% |
| StatementGRU | 11.2%   | 13.0%  | 8.4%      | 10.3% |
| ASTGRU   | 9.1%     | 10.2%  | 10.2%     | 10.3% |
| Devign   | 10.8%    | 13.1%  | 6.9%      | 9.6%  |
| **Average** | **9.9%** | **12.6%** | **9.0%** | **10.5%** |

Obviously, for our tested DL-based vulnerability detection models (i.e., TokenCNN, VulDeePecker, StatementGRU, ASTGRU, and Devign), the detection ability can be improved after using Humer to train them. For example, the F1 is only 45.8% when TokenCNN is trained in a completely random order, but it can be increased to 57.3% after applying Humer to train the model. Meanwhile, we also observe that when using Humer to train different models, the improved effectiveness is different. For instance, VulDeePecker’s F1 can increase up to 10.9% (62.8%-51.9%=10.9%), while Devign’s F1 can only increase

Fig. 6: The comparative results of DL-based vulnerability detectors (i.e., TokenCNN, VulDeePecker, StatementGRU, ASTGRU, and Devign) with and without Humer.
computing the difference between Baseline using Humer which are used to fine-tune the model to strengthen the detection ability of DL-based vulnerability detectors.

D. For Data Augmenter

In Humer, we leverage Data Augmenter to construct semantic-equivalent variants of those mispredicted samples which are used to fine-tune the model to strengthen the learning of misunderstood knowledge. Therefore, in this part, we conduct comparative experiments to evaluate Humer with and without Data Augmenter. Specifically, we construct an ablation experiment to achieve the goal, that is, only Data Augmenter is deleted. As shown in Figure 7, the red color represents the baseline result, the gray color represents the experimental result of Humer without Data Augmenter, and the pink color represents the experimental result of Humer with Data Augmenter.
experimental result of Humer without Data Augmenter, and the pink color represents the experimental result of Humer with Data Augmenter. To present the results more clearly, we use 0 and 1 to distinguish whether Data Augmenter is removed in Figure 7. For example, 3-5-0 means that the training set is first divided into three subsets to calculate the model-based difficulty. After sorting, the training set will be divided into five buckets, and then train a model by using Humer without Data Augmenter. 3-1 means that the training set is first sorted by their code complexity, and then is divided into three buckets to train a model by using Humer with Data Augmenter.

Through the results in Figure 7 we observe that whether Humer has Data Augmenter or not, after using Humer for training, the detection effectiveness of DL-based vulnerability detection models can be improved. But the overall effectiveness (i.e., F1) will increase more when Humer adopts Data Augmenter to fine-tune the model. In addition, we also find that there is a situation where after using Data Augmenter, the recall of the model may be higher but the precision is slightly lower, or the precision becomes higher but the recall becomes a little lower. For example, suppose that we first split the training set into five subsets to train five submodels for difficulty calculation, and then divide them into ten buckets for curriculum learning after sorting. At this point, if we use Humer without Data Augmenter (i.e., 5-10-0) to train Devign, the recall and precision are 64.9% and 57.6%, respectively. After adding Data Augmenter into Humer (i.e., 5-10-1), the recall can increase to 68.8% but the precision drops from 57.6% to 57.4%.

This phenomenon may correspond to the partial section phenomenon in human learning, that is, someone can do well in some subjects while poor in other ones. Suppose there is a student Tom who is more interested in mathematics than literature, so he is willing to spend more time learning mathematics. However, a person’s energy is limited. Since what he has been studying is mathematics knowledge, he may have forgotten part of literary knowledge while gaining more mathematics knowledge. Correspondingly, if there are many vulnerability samples in mispredicted data, most of the knowledge learned is the knowledge related to the vulnerability, which may cause the knowledge of some normal samples to be forgotten. In the end, more vulnerabilities will be detected but more normal samples will be marked as vulnerabilities at the same time, that is, recall becomes higher but precision becomes lower. To verify our guess, we compute the ratio of vulnerability samples to normal samples in Devign’s mispredicted data (i.e., 5-10-0 and 5-10-1). After the analysis, we find that more than 90% of the data are vulnerabilities, that is, the ratio of vulnerability samples to normal samples is more than 9:1. But in fact, the overall detection ability (i.e., F1) of these DL-based vulnerability detection models can be improved. In other words, models can learn more knowledge after using Data Augmenter than without using Data Augmenter.

E. Summary

Through our experimental results, we obtain several findings:
1) For Humer, model-based Difficulty Calculator is better than code-based Difficulty Calculator.
2) The training set is divided into different numbers of subsets (i.e., M) and buckets (i.e., N) can affect the detection effectiveness of Humer.
3) After training with Humer, the detection capabilities of DL-based vulnerability detection models can be improved.
4) For different DL-based vulnerability detection models, the enhancement effectiveness of using Humer is different.
5) Humer without Data Augmenter can also enhance the detection ability of DL-based vulnerability detection models.
6) Humer with Data Augmenter is better than Humer without Data Augmenter.

V. CASE STUDY

To further verify the practicability of Humer, we conduct a case study to check whether the use of Humer can help DL-based vulnerability detectors to discover new vulnerabilities from real-world open source software. We select seven widely used open source products as our test objects: Git [38], Htpdp [39], Libav [40], OpenSSL [41], Seamonkey [42], Thunderbird [43], and Xen [44]. The selected versions of these products include both several old versions and the latest version. In this way, we can report whether vulnerabilities in old versions have been “silently” patched in the latest version or not. As shown in Table V the total number of lines of code we analyzed exceeds 58 million lines.

### TABLE V: Summary of our tested open-source software

| Software | #Files | #Functions | #Lines of Code |
|----------|--------|------------|----------------|
| Git      | 1,427  | 24,384     | 844,049        |
| Htpdp    | 1,338  | 16,398     | 765,375        |
| Libav    | 4,983  | 37,682     | 2,087,350      |
| OpenSSL  | 3,195  | 24,512     | 120,658        |
| Seamonkey| 42,332 | 16,501     | 21,031,517     |
| Thunderbird| 51,139| 665,579    | 25,544,334     |
| Xen      | 14,829 | 207,447    | 8,279,792      |
| Total    | 119,243| 1,544,515  | 58,673,075     |

To commence our case study, we first use data in Table II (i.e., 12,460 vulnerabilities and 14,858 normal functions) to train five models (i.e., TokenCNN [12], VulDeePecker [11], StatementGRU [26], ASTGRU [27], and Devign [13]) with Humer. After training, we then leverage these five enhanced models to scan vulnerabilities from products in Table V. Through the results in Table III we see that the detection performance of TokenCNN, VulDeePecker, StatementGRU, ASTGRU, and Devign is not ideal when detecting real-world vulnerabilities. Therefore, to make our scanning results more accurate, we only consider a function as a vulnerability and report it as a warning when all five models label it as vulnerable.
TABLE VI: Vulnerabilities detected from Httpd product

| Target product | CVE ID       | Vulnerability release date | Vulnerable file in the target product | Patched or not in the latest version |
|----------------|-------------|---------------------------|--------------------------------------|--------------------------------------|
| Httpd-2.0.35  | CVE-2007-5000 | 12/13/2007 | modules/mappers/mod_imap.c | deleted in Httpd-2.4.51 |
|                | CVE-2011-4317 | 11/29/2011 | modules/mappers/mod_rewrite.c | patched in Httpd-2.4.51 |
|                | CVE-2013-5704 | 4/15/2014 | modules/loggers/mod_log_config.c | patched in Httpd-2.4.51 |
|                | CVE-2016-0718 | 5/26/2016 | srclib/apr-util/xml/expat/lib/xmlparse.c | deleted in Httpd-2.4.51 |
|                | CVE-2016-0718 | 5/26/2016 | srclib/apr-util/xml/expat/lib/xmltok_impl.c | deleted in Httpd-2.4.51 |
| Httpd-2.2.14  | CVE-2009-3555 | 11/9/2009 | modules/ssl/ssl_engine_io.c | patched in Httpd-2.4.51 |
|                | CVE-2009-3555 | 11/9/2009 | modules/ssl/ssl_engine_init.c | patched in Httpd-2.4.51 |
|                | CVE-2011-4317 | 11/29/2011 | modules/mappers/mod_rewrite.c | patched in Httpd-2.4.51 |
|                | CVE-2013-5704 | 4/15/2014 | modules/loggers/mod_log_config.c | patched in Httpd-2.4.51 |
|                | CVE-2014-0117 | 7/20/2014 | srclib/apr-util/xml/expat/lib/xmltok_impl.c | deleted in Httpd-2.4.51 |
|                | CVE-2016-0718 | 5/26/2016 | srclib/apr-util/xml/expat/lib/xmlparse.c | deleted in Httpd-2.4.51 |
|                | CVE-2016-0718 | 5/26/2016 | srclib/apr-util/xml/expat/lib/xmltok_impl.c | deleted in Httpd-2.4.51 |
| Httpd-2.4.51  | CVE-2014-0117 | 7/20/2014 | modules/proxy/proxy_util.c | Not patched |
|                | CVE-2017-7668 | 6/19/2017 | server/util.c | Not patched |

After scanning, we obtain a total of 343 warnings. To check if they are real vulnerabilities or false positives, we conduct manual analysis to compare them with some vulnerabilities\(^2\) in NVD one by one. If it is found that the two belong to the same pattern, it is judged as a real vulnerability. After analyzing all warnings, we find that 281 of them correspond to patterns of vulnerabilities in NVD, and the other 62 warnings are false positives.

Due to the limited page, we only present the scanning results from Httpd product in this section, the other detected vulnerabilities can be seen on our website\(^3\). Table VI shows the details including the corresponding CVE ID in NVD, the release date in NVD, the vulnerable file in the target product, and whether patched or not in the latest version of the target product. From the results in Table VI, we find that some vulnerabilities can exist in multiple versions, and some vulnerabilities may be patched or deleted in the latest version after being discovered. After statistical analysis, we observe that among 281 detected vulnerabilities, 98 of them have been patched by vendors in the latest version of corresponding products, 24 of them have been deleted, and the other 159 vulnerabilities still exist in the products. We have reported these vulnerabilities to their vendors and hope that they can release a patched version as soon as possible.

After collecting our detected vulnerability samples (i.e., 281 vulnerabilities), we also conduct another experiment to check how many vulnerabilities cannot be detected by the baseline models, that is, vulnerabilities that can only be detected after using Humer for training. Figure 8 presents the detected results of TokenCNN, VulDeePecker, StatementGRU, ASTGRU, and Devign without Humer. It shows that the use of Humer can make DL-based vulnerability detection models detect more vulnerabilities. In fact, after using Humer to train TokenCNN, VulDeePecker, StatementGRU, ASTGRU, and Devign, they can detect 52 (i.e., 281-229=52), 44 (i.e., 281-237=44), 40 (i.e., 281-241=40), 32 (i.e., 281-249=32), and 20 (i.e., 281-261=20) more vulnerabilities, respectively. In one word, using Humer can enable them to detect 37 (i.e., (52+44+40+32+20)/5=37.6) more vulnerabilities on average. Such results demonstrate the effectiveness of Humer on enhancing DL-based vulnerability detection models.

VI. DISCUSSIONS

As aforementioned, we select three different \(M\) to split the training set into different subsets for model-based difficulty calculation and three different \(N\) to divide the sorted training set into different buckets for curriculum learning. In fact, although the number of \(M\) and \(N\) are both three, we conduct a total of 24 experiments for each DL-based vulnerability detection model (Figure 7). In other words, the total number of experiments performed to evaluate Humer is 120 (i.e., \(24\times5=120\)) since we select five DL-based vulnerability detectors (i.e., TokenCNN \[17\], VulDeePecker \[11\], StatementGRU \[26\], ASTGRU \[27\], and Devign \[13\]). In our future work, we plan to buy more GPUs or spend more time to test more \(M\) and \(N\) to find more suitable results.

Because most of existing DL-based vulnerability detection methods are not open source, we only select five state-of-the-art models (i.e., TokenCNN, VulDeePecker, StatementGRU, ASTGRU, and Devign) to commence the evaluations of Humer. In fact, the code representations of these five methods are

\(^2\)We have collected some vulnerabilities with CVE-ID from NVD.

\(^3\)https://github.com/Humer-DL/Humer.
different, resulting in almost different neural networks used in the model. In this way, we can achieve more comprehensive evaluations to examine the effectiveness of Humer on enhancing DL-based vulnerability detection models. In the future, we will choose more DL-based vulnerability detection approaches to evaluate Humer.

From the results in Section IV.D, we see that in mispredicted data, if the ratio of vulnerability samples to normal samples is too different, the recall may increase but the precision may be slightly decreased. But in fact, the F1 can be still increased, that is, the overall performance is still improved. We plan to further analyze the semantic-equivalent variant samples to obtain the ratio of the vulnerability data to the normal data first. If the amount of a certain type of data is small, we will input this type of data into Data Augmenter and increase the number of code transformations and combinations to generate more of this type of data. By this, we can balance the ratio of vulnerability samples to normal samples to mitigate the issue.

In Data Augmenter, we design five code transformation rules to construct semantic-equivalent variant samples. In fact, a program may satisfy different rules. In this paper, we propose two types of transformations to generate the variants, that is, a simple transformation and a hard transformation. For simple transformation, we only apply one rule in one satisfied code fragment and other code remain unchanged. For hard transformation, we apply all satisfied rules to generate a complex variant. In our future work, we will implement more code transformation rules and design more different combinations of these rules to construct more semantic-equivalent variant samples of mispredicted data.

VII. RELATED WORK

A. Vulnerability Detection

Existing source code vulnerabilities detection methods can be classified into two main categories: the first is code-similarity-based approaches and the second is pattern-based techniques.

1) Code-similarity-based Vulnerability Detection

Code-similarity-based vulnerability detection methods apply different code similarity analysis algorithms to measure the similarity between known vulnerabilities and the program to be detected. In fact, the similarity can be evaluated from different perspectives, such as string-based [3], [4], tree-based [45], [8], token-based [46], [47], graph-based [6], and their hybrid-based [7]. Because the similarity is measured by analyzing known vulnerabilities and the source code to be detected, these code-similarity-based vulnerability detection tools are used to find cloned vulnerabilities.

2) Pattern-based Vulnerability Detection

To discover more vulnerabilities, researchers propose to extract the patterns of vulnerabilities and detect new vulnerabilities by matching vulnerability patterns. According to the degree of automation, these studies can be divided into three types. The first type [8], [9], [10] is to define vulnerability patterns by human experts. However, because experts cannot generate all patterns of different vulnerabilities, their detection effectiveness is not ideal. The second type [48], [49], [50], [51] is to extract certain features (e.g., API symbols [52]) first and then leverage some traditional machine learning algorithms (e.g., support vector machine) to detect vulnerabilities. The third type [11], [12], [13], [14], [15], [16], [17], [18], [53] is to use deep learning (DL) to detect vulnerabilities.

For example, VulDeePecker [11] first extracts the program slices of a program and then makes use of a bidirectional long short-term memory (BLSTM) to train a detector. muVulDeePecker [12] improves the implementation of VulDeePecker, it introduces the concept of code attention and uses it to help VulDeePecker complete multiclass vulnerability detection. Feng et al. design a tree-based vulnerability detector. They first apply static analysis to extract the abstract syntactic tree of programs and then apply preorder traversal search algorithm to convert the trees into sequences. Finally, these sequences will be used to train a bidirectional gated recurrent unit (BGRU) model to detect vulnerabilities. DeepWukong [53] distills the program semantics into a program dependency graph and splits it into several subgraphs according to the program points of interest. Then these subgraphs are fed into a graph neural network (GNN) to train a vulnerability detector.

B. Curriculum Learning

Inspired by children’s language learning, Elman [22] studies the influence of learning programs on synthetic grammar tasks. Through his experimental results, he finds that a recurrent neural network can learn a grammar when training data is presented from simple to complex order, but fails to do so when the order is random. In 2009, Bengio et al. [25] officially name this learning method as curriculum learning (CL) and verify that CL can not only improve the effectiveness but also speed the convergence of the model. Since then, CL has been widely used in various fields [54], [55], [56], [57], [58].

For example, El-Bouri et al. [55] combine CL with reinforcement learning to predict where in a hospital emergency patients will be admitted after being triaged. Guo et al. [56] use CL to help train deep neural networks on large-scale weakly-supervised web images. Tay et al. [57] introduce CL to tackle the problem of reading comprehension over long narratives. Zhou et al. [58] use uncertainty-aware CL in neural machine translation to improve the translation quality and convergence speed. Zhao et al. [59] propose a CL-based Deep QNetwork and introduce a teacher policy model to replace the traditional random sampling method, which significantly improves the effectiveness and stability of the dialogue tasks. Dai et al. [60] make use of a curriculum structure and a schema structure to improve the effectiveness of dialog state tracking. More related work can be found in a recent survey [61].

VIII. CONCLUSION

In this paper, we propose to train DL-based vulnerability detection models in a human-learning manner. Specifically, we implement a model-agnostic framework namely Humer, and select five state-of-the-art vulnerability detection methods (i.e., TokenCNN [17], VulDeePecker [11], StatementGRU...
D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, “Humer: A real-world vulnerability dataset. The experimental results indicate that using Humer can make them detect 12.6% more vulnerabilities on average. We also conduct a case study to demonstrate the practicability of Humer and finally discover 281 vulnerabilities that are not reported in NVD.

REFERENCES

[1] “Cves received.” https://nvd.nist.gov/vuln/search/statistics?form_type=BasicKerList&type=statistics&search_type=all&isCpeNameSearch=false 2021.
[2] “What is wannacry ransomware?” https://www.kaspersky.com/resource-center/threats/ransomware-wannacry 2021.
[3] J. Jang, A. Agrawal, and D. Brunley, “Redbug: Finding unpatched code clones in entire os distributions,” in Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P’12), 2012, pp. 48–62.
[4] S. Kim, S. Woo, H. Lee, and H. Oh, “Vuddy: A scalable approach for vulnerable code clone discovery,” in Proceedings of the 2017 IEEE Symposium on Security and Privacy (S&P’17), 2017, pp. 595–614.
[5] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, “Detection of recurring software vulnerabilities,” in Proceedings of the 2010 International Conference on Automated Software Engineering (ASE’10), 2010, pp. 447–456.
[6] J. Li and M. D. Ernst, “Chcd: Cloned buggy code detector,” in Proceedings of the 34th International Conference on Software Engineering (ICSE’12), 2012, pp. 310–320.
[7] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, “Vulpecker: An automated vulnerability detection system based on code similarity analysis,” in Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC’16), 2016, pp. 201–213.
[8] “Checkmarx.” https://www.checkmarx.com/ 2021.
[9] “Rough audit tool for security.” https://code.google.com/archive/p/rough-audit-tool-for-security/ 2021.
[10] “Flawfinder.” http://www.dwheeler.com/flawfinder/ 2021.
[11] Z. Li, D. Zou, S. Xu, X. Ou, and Y. Zhong, “Vuldeepecker: A deep learning-based system for vulnerability detection,” in Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS’18), 2018, pp. 1–15.
[12] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, “Vuldeepecker: A deep learning-based system for multiclass vulnerability detection,” IEEE Transactions on Dependable and Secure Computing, pp. 1–13, 2019.
[13] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, “Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks,” in Proceedings of the 2019 Advances in Neural Information Processing Systems (NIPS’19), 2019, pp. 10197–10207.
[14] G. Lin, J. Zhang, W. Luo, L. Pan, and Y. Xiang, “Poster: Vulnerability discovery with function representation learning from unlabelled projects,” in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS’17), 2017, pp. 2539–2541.
[15] X. Duan, J. Wu, S. Ji, Z. Rui, T. Luo, M. Yang, and Y. Wu, “Vulnsniper: Focus your attention to shoot fine-grained vulnerabilities,” in Proceedings of the 2019 International Joint Conference on Artificial Intelligence (IJCAI’19), 2019, pp. 4665–4673.
[16] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, “Sysev: A framework for using deep learning to detect software vulnerabilities,” IEEE Transactions on Dependable and Secure Computing, 2021.
[17] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, “Automated vulnerability detection in source code using deep representation learning,” in Proceedings of the 2018 IEEE International Conference on Learning and Applications (ICMLA’18), 2018, pp. 757–762.
[18] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang, “Combining graph-based learning with automated data collection for code vulnerability detection,” IEEE Transactions on Information Forensics and Security, vol. 16, pp. 1943–1958, 2020.
[19] “Software assurance reference dataset.” https://samate.nist.gov/SRD/in dex.php 2021.
[20] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, “Deep learning based vulnerability detection: Are we there yet?,” IEEE Transactions on Software Engineering, 2021.
[21] B. F. Skinner, “Reinforcement today,” American Psychologist, vol. 13, no. 3, p. 94, 1958.
[22] J. L. Elman, “Learning and development in neural networks: The importance of starting small,” Cognition, vol. 48, no. 1, pp. 71–99, 1993.
[23] G. B. Peterson, “A day of great illumination: Bf Skinner’s discovery of shaping,” Journal of the experimental analysis of behavior, vol. 82, no. 3, pp. 317–328, 2004.
[24] K. A. Krueger and P. Dayan, “Flexible shaping: How learning in small steps helps,” Cognition, vol. 110, no. 3, pp. 380–394, 2009.
[25] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, “Curriculum learning,” in Proceedings of the 26th annual international conference on machine learning, 2009, pp. 41–48.
[26] G. Lin, W. Xiao, J. Zhang, and Y. Xiang, “Deep learning-based vulnerability function detection: A benchmark,” in International Conference on Information and Communications Security. Springer, 2019, pp. 219–232.
[27] H. Feng, X. Fu, H. Sun, H. Wang, and Y. Zhang, “Efficient vulnerability detection based on abstract syntax tree and deep learning,” in IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). IEEE, 2020, pp. 722–727.
[28] D. Coleman, D. Ash, B. Lowther, and P. Oman, “Using metrics to evaluate software system maintainability,” Computer, vol. 27, no. 8, pp. 44–49, 1994.
[29] P. Oman and J. Hagemeister, “Metrics for assessing a software system’s maintainability,” in Proceedings of Conference on Software Maintenance 1992. IEEE Computer Society, 1992, pp. 337–338.
[30] I. Heidtger, T. Kupers, and A. Prakash, “A practical model for measuring maintainability,” in 6th international conference on the quality of information and communications technology (QUATIC 2007). IEEE, 2007, pp. 30–39.
[31] D. Coleman, B. Lowther, and P. Oman, “The application of software maintainability models in industrial software systems,” Journal of Systems and Software, vol. 29, no. 1, pp. 3–16, 1995.
[32] K. D. Welker, P. W. Oman, and G. G. Akinson, “Development and application of an automated source code maintainability index,” Journal of Software Maintenance: Research and Practice, vol. 9, no. 3, pp. 127–159, 1997.
[33] “A simple understanding of code complexity.” https://www.codegrip.tec h/productivity/a-simple-understanding-of-code-complexity/ 2021.
[34] T. J. McCabe, “A complexity measure,” IEEE Transactions on software Engineering, no. 4, pp. 308–320, 1976.
[35] M. H. Halstead, Elements of Software Science (Operating and programing systems series). Elsevier Science Inc., 1977.
[36] “Qemu.” https://www.qemu.org/ 2021.
[37] “Ffmpeg.” http://www.ffmpeg.org/ 2021.
[38] “Git.” https://git-scm.com/ 2021.
[39] “Httpd.” https://httpd.apache.org/ 2021.
[40] “Libav.” https://libav.org/ 2021.
[41] “Openssl.” https://www.openssl.org/ 2021.
[42] “Thunderbird.” https://www.thunderbird.net/ 2021.
[43] “Xen.” https://www.xenproject.org/xen-project-archives/ 2021.
[44] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “Deckard: Scalable and accurate tree-based detection of code clones,” in Proceedings of the 29th International Conference on Software Engineering (ICSE’07), 2007, pp. 96–105.
[45] H. Sajmani, V. Saini, J. Svaajlenko, C. K. Roy, and C. V. Lopes, “Sourceerc: Scaling code clone detection to big-code,” in Proceedings of the 38th International Conference on Software Engineering (ICSE’16), 2016, pp. 1157–1168.
[46] T. Kamiya, S. Kusumoto, and K. Inoue, “Cfinder: A multilingualic token-based code clone detection system for large scale source code,” IEEE Transactions on Software Engineering, vol. 28, no. 7, pp. 654–670, 2002.
[47] U. Shankar, K. Talwar, J. S. Foster, and D. A. Wagner, “Detecting format string vulnerabilities with type qualifiers,” in Proceedings of the 2001 USENIX Security Symposium (USENIX Security’01), 2001, pp. 201–220.
[48] M. Backes, B. Köpf, and A. Rybalchenko, “Automatic discovery and quantification of information leaks,” in Proceedings of the 2009 IEEE Symposium on Security and Privacy (S&P’09), 2009, pp. 141–153.
[49] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, “Automatic inference of search patterns for taint-style vulnerabilities,” in Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P’15), 2015, pp. 797–812.
[51] L. K. Shar, L. C. Briand, and H. B. K. Tan, “Web application vulnerability prediction using hybrid program analysis and machine learning,” IEEE Transactions on Dependable and Secure Computing, vol. 12, no. 6, pp. 688–707, 2014.

[52] F. Yamaguchi, M. Lottmann, and K. Rieck, “Generalized vulnerability extrapolation using abstract syntax trees,” in Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC’12), 2012, pp. 359–368.

[53] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, “Deepwukong: Statically detecting software vulnerabilities using deep graph neural network,” ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 30, no. 3, pp. 1–33, 2021.

[54] C. Florensa, D. Held, M. Wulfmeier, M. Zhang, and P. Abbeel, “Reverse curriculum generation for reinforcement learning,” in Conference on robot learning. PMLR, 2017, pp. 482–495.

[55] R. El-Bouri, D. Eyre, P. Watkinson, T. Zhu, and D. Clifton, “Student-teacher curriculum learning via reinforcement learning: predicting hospital inpatient admission location,” in International Conference on Machine Learning. PMLR, 2020, pp. 2848–2857.

[56] S. Guo, W. Huang, H. Zhang, C. Zhuang, D. Dong, M. R. Scott, and D. Huang, “Curriculumnet: Weakly supervised learning from large-scale web images,” in Proceedings of the European Conference on Computer Vision (ECCV), 2018, pp. 135–150.

[57] Y. Tay, S. Wang, L. A. Tuan, J. Fu, M. C. Phan, X. Yuan, J. Rao, S. C. Hui, and A. Zhang, “Simple and effective curriculum pointer-generator networks for reading comprehension over long narratives,” arXiv preprint arXiv:1905.10847, 2019.

[58] Y. Zhou, B. Yang, D. F. Wong, Y. Wan, and L. S. Chao, “Uncertainty-aware curriculum learning for neural machine translation,” in Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, 2020, pp. 6934–6944.

[59] Y. Zhao, Z. Wang, and Z. Huang, “Automatic curriculum learning with over-repetition penalty for dialogue policy learning,” in Proceedings of the AAAI Conference on Artificial Intelligence, vol. 35, no. 16, 2021, pp. 14540–14548.

[60] Y. Dai, H. Li, Y. Li, J. Sun, F. Huang, L. Si, and X. Zhu, “Preview, attend and review: Schema-aware curriculum learning for multi-domain dialog state tracking,” arXiv preprint arXiv:2106.00291, 2021.

[61] X. Wang, Y. Chen, and W. Zhu, “A survey on curriculum learning,” IEEE Transactions on Pattern Analysis and Machine Intelligence, 2021.