applied sciences

MDPI

Article

Instruction2vec: Efficient Preprocessor of Assembly Code to Detect Software Weakness with CNN †

Yongjun Lee [1], Hyun Kwon [2], Sang-Hoon Choi [3], Seung-Ho Lim [4], Sung Hoon Baek [5] and Ki-Woong Park [3,*]

1 Information Security at Graduate School of Information Security, Korea University, Seoul 02841, Korea; yjlee@formal.korea.ac.kr
2 School of Computing, Korea Advanced Institute of Science and Technology, Daejeon 34141, Korea; khkh@kaist.ac.kr
3 Department of Computer and Information Security, Sejong University, 209, Neungdong-ro, Gwangjin-gu, Seoul 05006, Korea; csh0052@gmail.com
4 Division of Computer and Electronic Systems Engineering, Hankuk University of Foreign Studies, Seoul 02450, Korea; slim@hufs.ac.kr
5 Department of Computer System Engineering, Jungwon University, Chungcheongbuk-do 28024, Korea; shbaek@jwu.ac.kr
* Correspondence: woongbak@sejong.ac.kr; Tel.: +82-2-6935-2453
† This paper is an extended version of our paper published in ICONI 2017.

Received: 2 September 2019; Accepted: 23 September 2019; Published: 30 September 2019

Abstract: Potential software weakness, which can lead to exploitable security vulnerabilities, continues to pose a risk to computer systems. According to Common Vulnerability and Exposures, 14,714 vulnerabilities were reported in 2017, more than twice the number reported in 2016. Automated vulnerability detection was recommended to efficiently detect vulnerabilities. Among detection techniques, static binary analysis detects software weakness based on existing patterns. In addition, it is based on existing patterns or rules, making it difficult to add and patch new rules whenever an unknown vulnerability is encountered. To overcome this limitation, we propose a new method—*Instruction2vec*—an improved static binary analysis technique using machine. Our framework consists of two steps: (1) it models assembly code efficiently using *Instruction2vec*, based on *Word2vec*; and (2) it learns the features of software weakness code using the feature extraction of Text-CNN without creating patterns or rules and detects new software weakness. We compared the preprocessing performance of three frameworks—*Instruction2vec*, *Word2vec*, and *Binary2img*—to assess the efficiency of *Instruction2vec*. We used the *Juliet Test Suite*, particularly the part related to Common Weakness Enumeration(CWE)-121, for training and Securely Taking On New Executable Software of Uncertain Provenance (STONESOUP) for testing. Experimental results show that the proposed scheme can detect software vulnerabilities with an accuracy of 91% of the assembly code.

Keywords: binary analysis; software weakness; convolutional neural network; *Word2vec*

1. Introduction

Potential software weakness that can lead to exploitable security vulnerabilities continues to pose a risk to computer systems. According to Common Vulnerability and Exposures (CVE) [1], 14,714 vulnerabilities were reported in 2017; this was more than twice the number reported in 2016. Various techniques have been proposed to detect vulnerabilities. Among them, static binary analysis detects vulnerabilities without executing binary code. Most static binary analysis processes generate a model by abstracting code and then match the generated model to an existing pattern or rule [2–4].

However, it is difficult to add new rules and patch them every time an unknown vulnerability arises. This limitation reduces the performance of static binary analysis and increases false positives.

In this paper, we propose an improved static binary analysis technique that automatically learns software weakness using machine learning to overcome the above-mentioned limitation. Our ultimate goal was to create a framework for auditing assembly code like humans. Our framework models code using *Instruction2vec* (https://github.com/firmcode/instruction2vec) and then learns and detects software weaknesses using Text-CNN [5]. Our study is the improvement over our earlier work [6], submitted to ICONI 2017, 17–20 December 2017, Vientiane, Laos. This paper inherits the contribution of the previous paper and proposes a more scalable framework. We also tested our framework with more datasets. Finally, we demonstrate that it can detect software weakness in real world code using STONESOUP. The differences between our framework and existing static binary analysis are described below.

Firstly, we suggest a new framework that models assembly code more efficiently. This is called *Instruction2vec*, inspired by *Word2vec* [7]. The modeling method of typical static binary analysis tends to over-approximate, which makes it difficult for vulnerabilities to be detected [2]. We studied code modeling method for machine to understand code like humans, taking a cue from *Word2vec* [5] of nature language processing (NLP). *Word2vec* vectorizes each word based on distributional semantics to make the machine understand the meaning of the word. As *Word2vec* cannot consider the syntax of the assembly language, we propose *Instruction2vec*, which can consider it. We proposed to effectively model the assembly code, and then compared it with *Word2vec* and *Binary2img* used in malware classification[8] to assess its performance.

Second, our framework can overcome the pattern-based limit of static binary analysis. Typical static binary analysis detects software weakness by modeling code and matching it with rules or patterns [2–4]. However, a pattern-based detection algorithm requires human resources and money because it uses the patterns or rules created by experts. In addition, it is difficult to resolve new software weakness using such an algorithm. Our approach performs static binary analysis using deep learning, which does not require patterns, rules, or designs from experts. Our framework can rapidly and easily respond to a new vulnerability by learning software weakness in a preformed deep learning model. This is because Text-CNN—which is our deep learning model—automatically learns the weakness code and extracts its features. The features of weakness code are depicted in a classification model, and Text-CNN can detect weakness in new code based on this model.

We compared *Instruction2vec* with three preprocessing frameworks—*Instruction2vec*, *Word2vec*, and *Binary2img*—to assess the efficiency of *Instruction2vec*. Our evaluation was focused on CWE-121, and, after training the *Juliet Test Suite* [9], we tested it with STONESOUP [10] (Securely Taking On New Executable Software of Uncertain Provenance). We compiled each source code to binary code and converted it into a dataset through preprocessing. The dataset was divided into training set and testing set. We tested *Instruction2vec*, *Word2vec*, *Binary2img* with the same dataset. We found that our framework could detect software weakness of assembly code and it recorded accuracy of 91.11%.

2. Background

Our framework is a combination of machine learning and static binary analysis, which can produce a great synergy when trained with a dataset of software weakness. We propose this framework to overcome the limitations of existing pattern-based analysis. The pattern-based approach of static binary analysis cannot handle increasing vulnerabilities rapidly. Our framework can improve performance by learning from an increasing number of datasets, whereas existing static binary analysis provides the same performance regardless of dataset size.

2.1. Limitations of Static Binary Analysis

Static binary analysis is a technique for analyzing binary code without the execution or knowledge of source code. The first step in most static binary analysis methods is the modeling of assembly

code, which is the conversion of binary code into an intermediate language or abstracting code. The purpose of modeling is to produced graphs for various features such as control-flow, data-flow, and control-dependence graphs. The graphs drawn obtained from modeling, are compared with the observed pattern of vulnerabilities. If a graph matches the pattern, it is detected as a vulnerability.

Our framework is different from typical static binary analysis in two aspects. First, machine learning is applied to our framework instead of algorithmic modeling. The modeling method of typical static binary analysis tends to over-approximate, i.e., existing code information can be distorted in the process of drawing graphs. In contrast, *Instruction2vec* is optimized to extract the features of software weakness. Second, our framework does not use predefined patterns. Generally, patterns are created by experts, and this requires considerable time, effort, and cost. Our framework is low-cost and efficient because it automatically learns datasets using Text-CNN and creates a classification model.

2.2. Advantages of Machine Learning

Machine learning has been used in various fields, particularly in computer science, statistics, and data mining, because it provides significant advantages. According to Mitchell [11] "the more we learn, the better we can expect the program to perform". Hence, our framework can provide better performance as it learns more software weakness datasets. This feature makes our framework different from existing pattern-based algorithms. Our approach is to maximize this advantage by applying it to static binary analysis. In this section we describe the machine learning employed in this study. Our framework consists of *Instruction2vec* for code modeling and Text-CNN for learning. First, we explain *Word2vec*, followed by Text-CNN.

2.2.1. *Word2vec*

Various methods that enable machines to understand words have been used in Natural Language Processing (NLP). One of these is *Word2vec* [7], which is an algorithm that transforms words into vectors. The reason for converting text to a vector is to make a machine understand its meaning more accurately. Hence, words with similar meanings end up laying close to each other . For example, if we know the vectors of king, man, and woman, then we can infer the vector of queen as follows:

$$\overrightarrow{king} - \overrightarrow{man} + \overrightarrow{woman} = \overrightarrow{queen}.$$

Word2vec is a two-layer neural net that processes text. Its input is a text corpus, and its output is a set of vectors. *Word2vec* contains two distinct models, i.e., a continuous bag of words and a skip-gram. The model learns to map each discrete word ID, which ranges from 0 to the number of words in the vocabulary, into a low-dimensional continuous vector space from their distributional properties observed in a text corpus. We propose *Instruction2vec* based on the characteristics of *Word2vec*.

2.2.2. Convolutional Neural Networks (CNNs) and Text-CNN

Convolutional neural networks (CNNs) [12] are a type of deep learning algorithm with extremely high accuracy in image classification problems. Generally, a CNN consists of an input layer, an output layer, and multiple hidden layers with convolutional, pooling, or fully connected layers. Convolutional layers apply a convolution operation to the input using learnable filters. The operation slides each filter across the width and height of the input volume and computes the dot products between the entries of the filter and the input at any position. Unlike fully connected feedforward neural networks, which generate a high number of neurons for shallow architecture, a CNN reduces the number of free parameters, enabling the network to become deeper with fewer parameters. Pooling layers operate the outputs of neuron clusters at one layer into a single neuron in the next layer using the max or average operation. The function of pooling layers is to progressively reduce the spatial size of the representation to reduce the number of parameters and computation in the network and hence control overfitting. Fully connected layers connect every neuron in one layer to every neuron in another

layer, as seen in a multilayer perceptron neural network. All activations in the previous layer can be computed with matrix multiplication followed by bias offset.

In recent years, CNNs have been used in NLP beyond the field of image recognition. Text-CNN trains a CNN with one layer of convolution on top of word vectors obtained from an unsupervised neural language model. Word vectors are essentially feature extractors that encode the semantic features of words in their dimensions. In the present work, we use the Text-CNN that contains fine-tuning hyperactive parameters for our task. Despite the minor tuning of hyperactive parameters, this model achieves superlative performance in software weakness detection.

2.3. Software Weakness Dataset

Software weakness is learned using machine learning. Therefore, we require a large amount of software weakness datasets. Software weakness datasets have been created in various forms (CVE, CWE [13], and SARD [14]) in the past, and new datasets are being generated. We use the Juliet Test Suite in this study, which is an authorized dataset that is publicly accessible.

Juliet Test Suite is a collection of vulnerable datasets created by the National Institute of Standards and Technology (NIST). These datasets are categorized by CWE, and the archives are referred to as the *Juliet Test Suite*. It contains examples organized under 118 different CWEs. It provides several types of vulnerable code (C/C+, Java). The code is categorized into good and bad cases to make it suitable for supervised learning. Furthermore, as most cases are concise, they can be regarded as data without noise. In this work, we compile the *Juliet Test Suite* source code and process it into a dataset suitable for the CNN.

STONESOUP was created by the Intelligence Advanced Research Projects Activity (IARPA), specifically for testing static analysis tools. STONESOUP is a collection of C and Java testcases based on 16 widely-used open-source software packages in which vulnerabilities have been seeded. We use STONESOUP for testing and not for learning because it is more complex than the *Juliet Test Suite*, and hence it is difficult to detect vulnerabilities using STONESOUP. We validated our framework using the testcase in "STONESOUP Phase 1–Memory Corruption for C".

3. Related Work

3.1. Vulnerability Detection Using Machine Learning

In 2011, Yamaguchi published a study on vulnerability detection using machine learning in "Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities Using Machine Learning" [15]. In the paper, Yamaguchi identified vulnerabilities using a feature based on API usage pattern vectors. However, the approach has the limitations that it is white-box based and only API functions are included in the feature. Grieco introduced a new approach in "Toward large-scale vulnerability discovery using machine learning" [16] in 2016. The method extractsd static and dynamic features from a black box to overcome the limitation of the previous approach. Furthermore, when extracting features, *Word2vec* and a bag of words are used to vectorize API functions and argument values. Even though this method is improved by extracting dynamic features, it still has significant limitations because only the features based on API functions and argument values are used.

Several papers about deep-learning-based vulnerability detection were published in 2018. Li published a paper about vulnerability detection based on deep learning. In "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection" [17], VulDeePecker generated code gadgets, transformed them into vectors, and then detects vulnerabilities using Bidirectional LSTM(BLSTM). It was found that deep learning provided higher accuracy compared to pattern-based and code-similarity-based vulnerability detection systems. Russell proposed a detection technique using CNNs in "Automated Vulnerability Detection in Source Code Using Deep Representation Learning"[18]. They embedded source code using their lexer and employed convolutional feature extraction. They utilized the random forest classifier to classify vulnerabilities. However, the

above-mentioned methods commonly use general source code, not assembly code, and they are provided in API functions. Thus, there are limitations in expressing the structure of a program in detail.

In this paper, we propose a method to detect software weaknesses by extracting features from assembly code and using deep learning to overcome the above-mentioned limitations. We can expect reasonably higher accuracy because assembly code describes the structure of a program in more detail compared to API functions.

3.2. Malware Classification via Binary Data Visualization

Research has been continuously conducted to visualize binary data and to derive meaningful results. In particular, malicious code analysis techniques that require binary analysis are being developed. In 2011, Nataraj proposed a method to visualize and classify malicious code files [8]. He visualized malicious binary code by converting it into a binary 8-bit vector and then changing it to the grayscale format. In this manner, binary sections (text, rodata, data, etc.) could be easily distinguished when visualizing binary code, and it could be efficiently used for image classification. For example, in the case of various malicious codes belonging to the same family, a few sections were modulated, but the overall binary image was characterized by a large similarity to the images belonging to the same family. The binaries of eight malicious codes were visualized and categorized into K-NN. This enabled the family to be classified with an accuracy of 98%.

In 2015, Microsoft hosted a challenge [19] on how to use Kaggle to disclose data from its vaccine server and analyze publicly available malware samples to improve the accuracy of nine family classifications. As a result of the challenge, the visualization of malicious code binaries and family classification were ranked with the highest accuracy. Mansour Ahmadi developed the most accurate classification method in the challenge.

The study classified nine types of malicious codes with 99.8% accuracy by hybridizing the methods of the binary visualization of malicious codes and learning methods such as symbol, metadata, and entropy. In this study, we use the *Binary2img* technique to classify binary data using the vulnerable code registered in the CWE list.

4. Proposed Scheme

Our goal was to detect software weakness based on the deep learning of binary code. Hence, we apply various approaches for embedding assembly code effectively based on the CNN. We propose three frameworks and compare their accuracy through tests. The first framework uses *Instruction2vec*, which is an improved version of *Word2vec*. The second and third frameworks utilize *Word2vec* and *Binary2img*, respectively. All frameworks operate in a similar manner, with the exception of preprocessing. We train each framework using datasets and test them with the trained model.

Figure 1 shows the training process begins by preprocessing the labeled ELF (executable and linkable format) files. Each preprocessing methods (*Word2vec*, *Instruction2vec*, or *Binary2img*) generate as a dataset by preprocessing the ELF files that are labeled as software weakness or not (Good or Bad case). Deep learning models extract the features of data during the learning process and create predictive models. Figure 2 shows that the testing process is to classify new ELF files with software weakness. The ELF files in the testing process are unlabeled datasets. The first step in testing is to preprocess the ELF files to create a dataset. This is the same as the first step of the training. The dataset is passed to the predictive models created in the training process. The predictive models classify the dataset based on the features extracted during training.

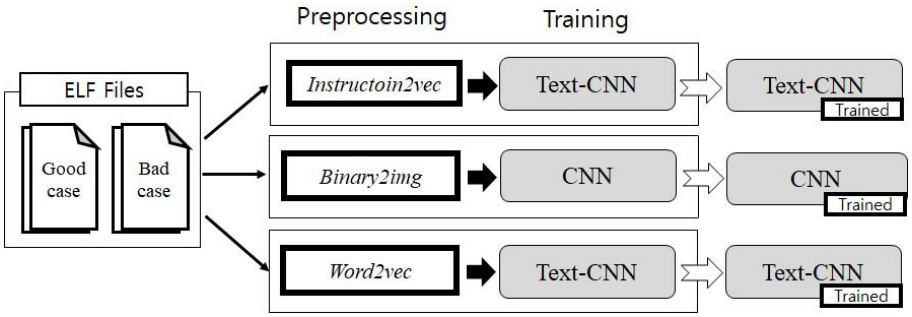

Figure 1. Framework design for training.

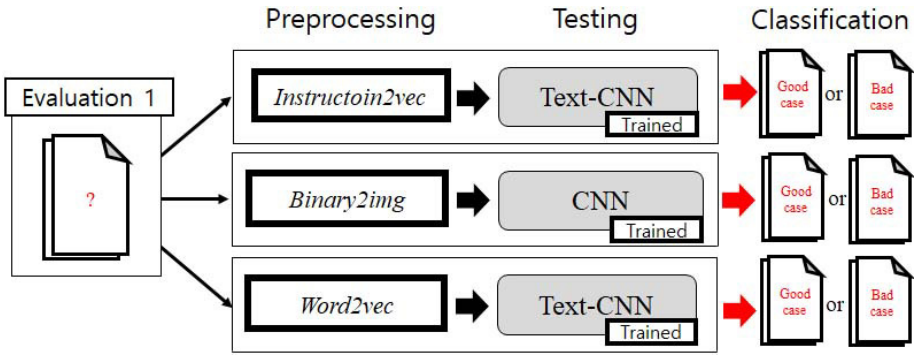

Figure 2. Framework design for testing.

4.1. Design of Instruction2vec

The purpose of *Instruction2vec* is to vectorize the instructions of the assembly code. We assert that deep learning should learn assembly code before learning software weaknesses. To appropriately learn the assembly code in the deep learning model, we require effective preprocessing. We preceded the approach using *Word2vec* in Section 4.3; it does not achieve the good accuracy because *Word2vec* can't consider about the syntax of the assembly language. Hence, we propose *Instruction2vec* which can consider about it for improvement accuracy.

Figure 3 shows the overall process of *Instruction2vec*. First, *Word2vec* generates a lookup table by reading the entire code. In this step, we construct a set of words through an appropriate morpheme analyzer for the assembly code, which extracts library functions, opcodes, registers, and hex values from the instruction and regards them as words. This set of words has a much smaller amount than the set of words used in the NLP. This characteristic means that *Word2vec* can learn effectively by reducing the vector size used in words. Second, the parser reads the assembly code and constructs the appropriate dimension for the syntax. Assembly code mostly has a fixed syntax, which means it uses one opcode and two operands (with four values). We could create a fixed dimension with a total of nine digits through one opcode and eight (4×2) operands. The reason for creating a fixed dimension is that Text-CNN must receive a fixed input size. Third, each opcode and operands in the dimension are referenced in the lookup table and fill into a vector. If the value of the operand is less than 4, it is filled with 0. The vector size is variable depending on the implementation of the lookup table. Since each vector value is contained in nine places, the dimension size of one instruction is "vector size $\times$ 9".

Through the above process we were able to obtain a fixed dimension in a single-line instruction. We use the above process to concatenate N dimensions to obtain a fixed dimension of N rows of instructions. Since *Instruction2vec* uses Text-CNN, N size should also be fixed. In the case of Section 5.1, we set N suitable for learning *Juliet test Suite* (Details are given in Section 5.1). If the size of the code to be analyzed is longer than N size, we split the code as in Section 5.2 to construct a set of instructions.

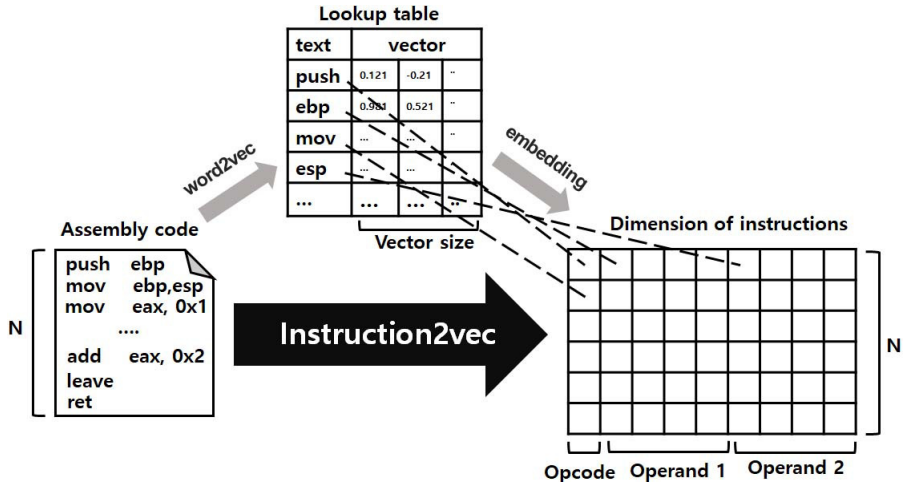

Figure 3. *Instruction2vec* scheme.

The sequence of instructions to concatenate is one of the factors that have a great effect on the learning results. Since *Instruction2vec* uses Text-CNN based on NLP, *Instruction2vec* tried to place instructions similar to the order in which people read code. The first instruction is the prologue of the function to be analyzed, and it places the instruction from top to bottom in order until the function ends. If the function calls function A, place the code for function A in the middle. When function A ends, it returns to the original function and continues to place. This sequence is similar to the execution sequence, but has the difference that, when branching is encountered, all the branches are analyzed, not just one side. Through the above procedure, *Instruction2vec* can construct the dimension of "N $\times$ vector size $\times$ 9" by concatenating N lines of instructions. This dimension is learned using Text-CNN. As a result, Text-CNN receives the syntax of the assembly code without distortion.

4.2. Design of Binary2img

Binary2img is a technique of embedding binary code in terms of data visualization. That is, it embeds binary code into images and lets a CNN learn the images. This method is different from *Instruction2vec*, which embeds assembly code into words. A few studies have applied this approach to malware classification and achieved considerably high accuracy. We apply this approach to static binary analysis.

The scheme of encoding a binary file to an image is shown in Figure 4. We apply a method to the image data of ELF binary files using red, green, and blue colors, which are referred to as three channels. We read 24 bits (true color) of an ELF binary file to represent the three channels as one pixel. To express the read value in pixels, we change the binary of 1 to 8 bits to integer, calculate the integer as mod 256, and set the value derived from mod 256 to red. Similarly, we read the binary of 9 to 16 bits, change the binary value to integer, calculate the integer as mode 256, and set the value derived from mod 256 to green. We read the binary of 17 to 24 bits and obtain the blue value through the same process. In the process, the size of an image is fixed as 256 $\times$ 256 pixels. The image size in the training data does not exceed 64 $\times$ 64 pixels. Finally, if the binary does not exist, the value of the RGB parameter is fixed at padding (255).

Binary2img has two differences compared to *Instruction2vec*. First, *Binary2img* embeds the assembly code as an image. This can be compressed to a very small size compared to *Instruction2vec*, but it cannot reflect the syntactic characteristics of the assembly code at all. Second, *Binary2img* does not consider the execution flow of code. *Instruction2vec* generates a dimension considering the execution flow, but *Binary2img* embeds the code into the image regardless of the execution order of the code. *Binary2img* may embeds two pieces of code with completely different execution sequences into similar images. This means loss of information about the code and adversely affect the learning results.

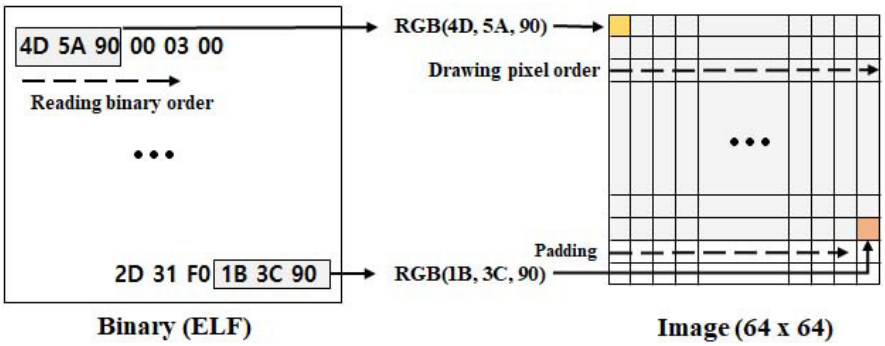

Figure 4. Binary2vec scheme.

4.3. Design of Word2vec

Word2vec is one of the methods of vectorizing words. We use *Word2vec* to efficiently vectorize assembly code. Our approach is to slice assembly code into each instruction and regard it as one word. For example, "push', "ebp", and "esp" are treated as a single word. *Word2vec* reads all words, vectorizes them, and constructs a lookup table. Finally, it concatenates the words by referring to the lookup table and creates the dimension. This is the most commonly used method in NLP, and it shows excellent performance in sentence classification. However, this method is not appropriate for software weakness classification because it does not consider the syntax of assembly code.

5. Experiment and Evaluation

An evaluation was performed to assess whether CNN models, which learn software weakness codes, can identify the weakness of new codes. In this study, we conducted two evaluations with three frameworks, as shown in Figure 5. In both evaluations we used a model that learns Juliet Test Suite codes. In the first and second evaluations, we tested the frameworks with the *Juliet Test Suite* and STONESOUP codes, respectively.

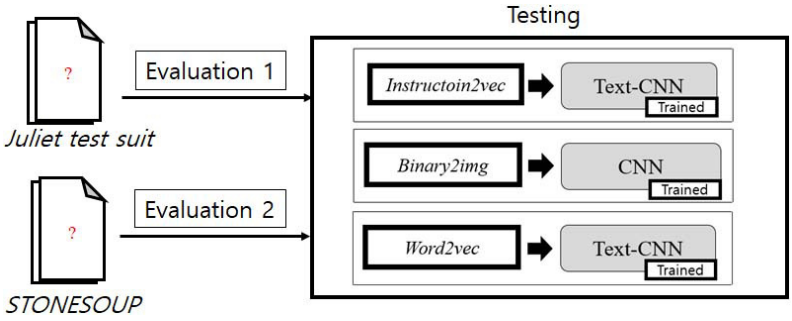

Figure 5. Process of evaluations.

5.1. Evaluation 1–Juliet Test Suite

In Evaluation 1, the three frameworks learned the datasets generated from the Juliet test suite, and they were tested with the same datasets. The frameworks adopted the same ELF file and environment, and only preprocessing and training were different. The overall evaluation process was as follows: First, a dataset was created from the ELF file using each preprocessor (*Instruction2vec*, *Binary2img*, and *Word2vec*). Second, we used 70% of the dataset for training and 30% for testing. There was no overlap between the training and testing datasets. Third, each framework was trained with the training dataset. Last, the frameworks that completed learning were tested with the testing dataset.

5.1.1. Dataset Generation

The *Juliet Test Suite* is a certified code sample created by NIST to evaluate static tools. We created datasets by utilizing the following three characteristics of the *Juliet Test Suite*: First, as the example codes of the *Juliet Test Suite* were categorized by CWE, they contributed to efficient training. Second, the codes were labeled as "good case" or "bad case". Two kinds of labeled codes were suitable for the binary classification model of the CNN. Finally, the codes were extremely concise.

We selected "CWE-121: Stack-based Buffer Overflow" among several CWEs for our evaluation because stack-based buffer overflow may or may not be a weakness owing to a single instruction line. All good cases and bad cases for each weakness were considerably similar. The good and bad cases with a large difference in length were excluded. We used the same number of datasets for good and bad cases for effective training. Each type of case contained 4599 codes, providing a total of 9198 codes. Codes were compiled with 32-bit GCC, and symbol table deletion and position-independent executables were applied. We generated the dataset through the preprocessor of each framework from the ELF files obtained through compilation. Out of 9198 datasets, 6439 were considered as the training dataset and the remaining 2759 as the testing dataset.

The datasets preprocessed by *Binary2img* may have contained information other than text sections, such as data sections. In addition, a CNN model with these datasets can classify codes using surrounding information (symbols table, data section, etc.). Therefore, only text sections were visualized in the dataset of *Binary2img* in ELF files.

5.1.2. Environment

We constructed three CNN models and efficiently trained them using the datasets created through preprocessing. Table 1 shows the models for *Instruction2vec*, *Binary2img*, and *Word2vec*, respectively. The first model was Text-CNN, which was trained using the datasets preprocessed with *Instruction2vec*. The input of Text-CNN was 117 $\times$ 320 dimensions, i.e., it could accept 320 lines of instruction and each instruction was expressed in 117 dimensions. The sizes of filters were 2, 4, 6, 8, 10, 12, 14, and 16, and the number of each filter was 150. The second model was the general CNN, which was trained using the datasets preprocessed with *Binary2img*. This model consisted of two convolution layers, two max pooling layers, and a fully connected layer. The third model used in *Word2vec* was similar to Text-CNN in *Instruction2vec*. The input of Text-CNN was 1,100 $\times$ 13 dimensions, and filter sizes were 4, 10, 20, and 60. We constructed three CNN models to efficiently train them using datasets created through preprocessing. The first model was Text-CNN to be trained in datasets preprocessed with *Instruction2vec*. The input of it was 117 $\times$ 320 dimensions: it could accept 320 lines of instruction and each instruction was expressed in 117 dimensions. The sizes of filters were 2, 4, 6, 8, 10, 12, 14, and 16, respectively, and the number of each filter was 150. The second model was General CNN to be trained in datasets preprocessed with *Binary2img*. This model consisted of two convolution layers, two max pooling layers, and a fully connected layer. The third model used in *Word2vec* was similar to Text-CNN in *Instruction2vec*. The input of Text-CNN was 1100 $\times$ 13 dimensions, and the filter size was 4, 10, 20, and 60, respectively.

Every model was constructed using TensorFlow version 1.7. The models were trained using CUDA version 8.1 and a TITAN XP GPU. We used a computer with an Intel i5-8600k CPU, 16 GB RAM, and Windows 10 operation system.

Table 1. Parameters of models.

Model Type	*Instruction2vec* Text-CNN	*Binary2img* Vanilla CNN	*Word2vec* Text-CNN
Input	117 × 320	64 × 64	13 × 1100
Filters	Conv [117, 2, 128] Conv [117, 4, 128] Conv [117, 6, 128] Conv [117, 8, 128] Conv [117, 10, 128] Conv [117, 12, 128] Conv [117, 14, 128] Conv [117, 16, 128]	Conv [3, 3, 32] Max pooling [2, 2] Conv [3, 3, 64] Max pooling [2, 2]	Conv [13, 4, 128] Conv [13, 10, 128] Conv [13, 20, 128] Conv [13, 60, 128]
Dropout	0.4	0.5	0.4
Learning rate	0.001	0.001	0.001
Initializer	xavier initializer	-	xavier initializer
Optimizer		AdamOptimizer	
Batch size		128	
Epochs		550	

5.1.3. Results

Figures 6–8 represent accuracy, recall, and precision, respectively, of each experiment. The process with the highest accuracy, 97.53%, was *Binary2img* (Table 2). However, the accuracy of other frameworks was also over 96%, which did not indicate a significant difference. Recall was the true positive rate, which means the ratio of the bad case actually found in the entire bad case (Recall = TP/(TP + FN)). The recall of *Instruction2vec* was 97.07%, indicating that *Instruction2vec* detected almost all bad cases. Finally, precision refers to the rate that a detected bad case was actually a bad case (Precision=TP/(TP + FP)). Precision is in inverse proportion to false-positive. Both *Binary2img* and *Instruction2vec* had precision of over 96% and can be considered as having very low false positives.

Table 2. Results of Evaluation 1.

Description	Accuracy	Recall	Precision
Instruction2vec	96.81%	97.07%	96.65%
Binary2img	97.53%	97.05%	97.91%
Word2vec	96.01%	96.07%	95.92%

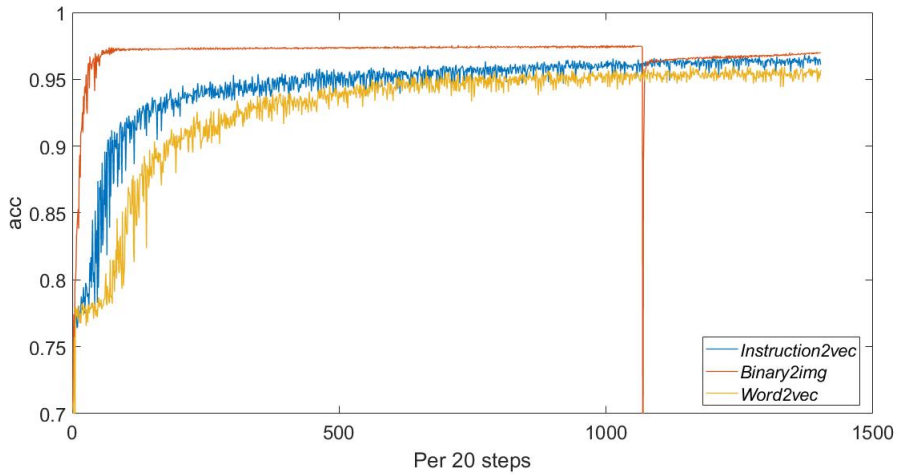

Figure 6. Result graph of accuracy.

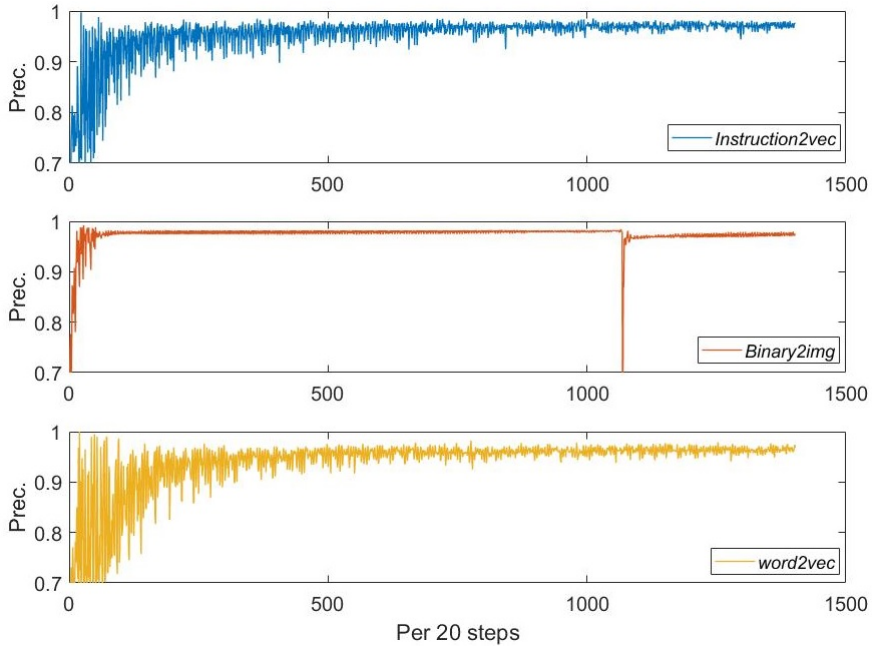

Figure 7. Result graph of precision.

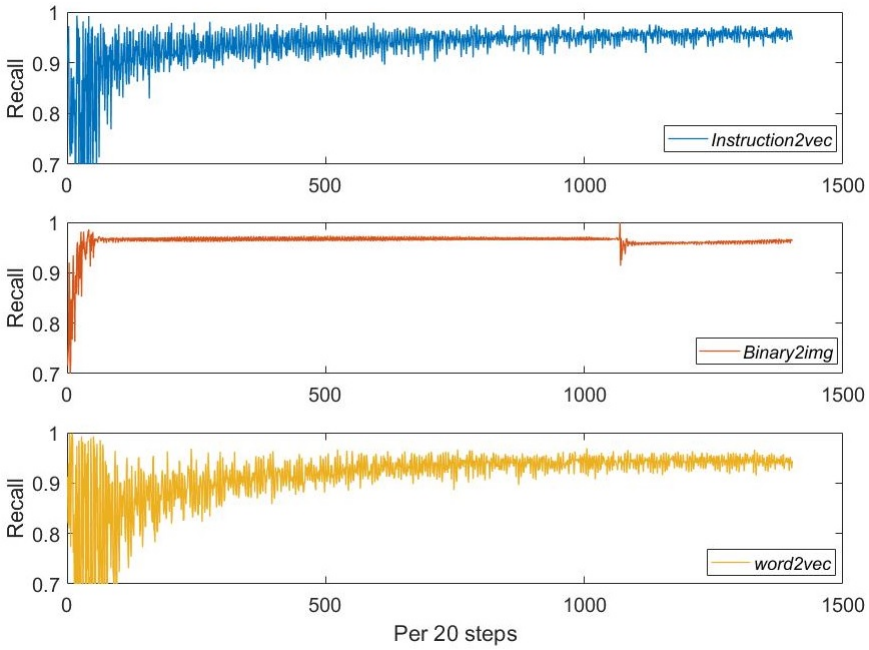

Figure 8. Result graph of recall.

Through Evaluation 1, we showed that deep learning models can learn and classify software weakness. However, each model exhibited similar accuracy. Thus, we cannot determine which model was better. This could be because the codes of the *Juliet Test Suite* are too simple and similar. Deep learning models can easily classify these codes. Therefore, tests needed to be performed with more complex codes to determine which framework was better. In Evaluation 2, we conducted experiments on the same models using more complex test codes.

5.2. Evaluation 2–STONESOUP

We used the models trained in Evaluation 1. The experimental process was the same as Evaluation 1, and only the step of adjusting the length when creating the dataset was added. We chose STONESOUP of NIST for testing. STONESOUP testcase was created by injecting software weakness into a particular base program, and we could achieve our goal of Evaluation 2 using these cases. We got example codes of STONESOUP for "CWE-121: Stack-based Buffer Overflow", which we used for training, and compiled them to ELF files. Then, the ELF files were preprocessed through each framework.

5.2.1. Dataset Generation

We used the testcase of "IARPA STONESOUP Phase 1–Memory Corruption for C" as the datasets for this evaluation. We selected 15 source code samples for "CWE-121: Stack-based Buffer Overflow". In the preprocessing of *Instruction2vec* and *Word2vec*, one ELF file was split into several assembly code files. This was because the maximum length of one file was fixed. Among the separate assembly code files, only the parts with vulnerabilities were labeled as "bad cases". We created 15 ELF files from STONESOUP and generated 79 datasets, which were divided into 15 bad cases and 64 good cases. Finally, they were vectorized by preprocessors and the datasets were created. Text-CNN classified these datasets. As a result, we could determine the exact location of weakness code in the ELF file.

In the preprocessing of *Binary2img*, we modified the code directly to balance good and bad cases to perform a fair test. We removed the annotated part of the testcase, which contained software weakness, and used it as a good case. As a result, we obtained 30 codes with a 1:1 ratio of good cases to bad cases. We compiled these codes into ELF files and visualized them at a size of 400 $\times$ 400 pixels. Then, we resized the files to 64 $\times$ 64 pixels size and used them for testing.

5.2.2. Result

In this evaluation, the tests with *Instruction2vec* datasets recorded a relatively high accuracy of 91.11%, (Table 3). A total of 14 out of 15 weaknesses in the datasets were detected, and recall was 93.33%. The framework that used *Instruction2vec* could detect software weakness in actual programs. In contrast, the test results obtained from other frameworks were unsatisfactory. The accuracy of *Binary2img* and *Word2vec* was less than 53%. This showed that these two frameworks were not suitable for detecting software weakness in actual programs.

Table 3. Total of result evaluation-2.

Description	Accuracy	Loss	Recall	Precision
Instruction2vec	91.11%	0.3058	93.33%	70.00%
Binary2img	53.33%	0.6894	46.39%	49.67%
Word2vec	18.98%	4.6809	100%	18.98%

6. Conclusions

We proposed a software weakness detection framework using machine learning. Our framework efficiently models assembly code using *Instruction2vec* and learns the features of software weakness code using Text-CNN. We showed that our framework could detect software weaknesses without patterns or rules. In addition, it showed potential for detecting software weakness in actual programs. *Instruction2vec* has features that can be used in other neural net models. In future, we will apply the features to neural net algorithms other than CNNs. A neural net will not only understand code but also understand and learn the overall flow of a program. In addition, it will provide additional benefits if it is combined with a static analysis technique or a dynamic analysis technique.

Author Contributions: Conceptualization, investigation, writing—original draft preparation, visualization, validation, Y.L. and S.-H.C.; methodology, software, Y.L. and S.-H.L.; writing—review and editing, H.K.; supervision, K.-W.P. and H.K.; project administration, funding acquisition, K.-W.P., S.-H.L., S.H.B.

Funding: This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2018-0-00420) and supported by the National Research Foundation of Korea (NRF) (NRF-2017R1C1B2003957).

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

References

1. Common Vulnerabilities and Exposures. Available online: https://cve.mitre.org/ (accessed on 20 September 2019).
2. Shoshitaishvili, Y.; Wang, R.; Salls, C.; Stephens, N.; Polino, M.; Dutcher, A.; Grosen, J.; Feng, S.; Hauser, C.; Kruegel, C.; et al. Sok: (State of) the art of war: Offensive techniques in binary analysis. In Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2016; pp. 138–157.
3. Brooks, T.N. Survey of Automated Vulnerability Detection and Exploit Generation Techniques in Cyber Reasoning Systems. 2017. Volume abs/1702.06162. Available online: http://xxx.lanl.gov/abs/1702.06162 (accessed on 20 September 2019).
4. Pewny, J.; Garmany, B.; Gawlik, R.; Rossow, C.; Holz, T. Cross-architecture bug search in binary executables. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015; pp. 709–724.
5. Kim, Y. Convolutional neural networks for sentence classification. *arXiv* **2014**, arXiv:1408.5882.
6. Lee, Y.J.; Choi, S.H.; Kim, C.; Lim, S.H.; Park, K.W. Learning binary code with deep learning to detect software weakness. In Proceedings of the KSII The 9th International Conference on Internet (ICONI) 2017 Symposium, Vientien, Laos, 17–20 December 2017.
7. Goldberg, Y.; Levy, O. word2vec Explained: Deriving Mikolov et al.'s negative-sampling word-embedding method. *arXiv* **2014**, arXiv:1402.3722.
8. Nataraj, L.; Karthikeyan, S.; Jacob, G.; Manjunath, B. Malware images: Visualization and automatic classification. In Proceedings of the 8th International Symposium on Visualization for Cyber Security, Pittsburgh, PA, USA, 20 July 2011; p. 4.
9. Boland, T.; Black, P.E. Juliet 1. 1 C/C++ and java test suite. *Computer* **2012**, *45*, 88–90. [CrossRef]
10. Vanderlinde, W. Securely Taking on New Executable Software of Uncertain Provenance (STONESOUP). Available online: http://www.iarpa.gov/index.php/research-programs/stonesoup (accessed on 20 September 2019).
11. Mitchell, T.M. Machine learning and data mining. *Commun. ACM* **1999**, *42*, 30–30. [CrossRef]
12. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. ImageNet Classification with Deep Convolutional Neural Networks. In Proceedings of the 25th International Conference on Neural Information Processing Systems—Volume 1, Lake Tahoe, NV, USA, 3–6 December 2012; NIPS'12, pp. 1097–1105.
13. Martin, R.A. *Common Weakness Enumeration*; Mitre Corporation: McLean, VA, USA, 2007.
14. Black, P.E. SARD: Thousands of reference programs for software assurance. *J. Cyber Secur. Inf. Syst. Tools Test. Tech. Assur. Softw. Dod Softw. Assur. Community Pract.* **2017**, *2*, 5.
15. Yamaguchi, F.; Lindner, F.; Rieck, K. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In Proceedings of the 5th USENIX Conference on Offensive Technologies, San Francisco, CA, USA, 8 August 2011; USENIX Association: Berkeley, CA, USA, 2011; WOOT'11, p. 13.
16. Grieco, G.;Grinblat, G.L.; Uzal, L.; Rawat, S.; Feist, J.; Mounier, L. Toward large-scale vulnerability discovery using machine learning. In Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, New Orleans, LA, USA, 9–11 March 2016; pp. 85–96. [CrossRef]
17. Li, Z.; Zou, D.; Xu, S.; Ou, X.; Jin, H.; Wang, S.; Deng, Z.; Zhong, Y. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. *CoRR* 2018. Available online: http://xxx.lanl.gov/abs/1801.01681 (accessed on 20 September 2019).

18. Russell, R.L.; Kim, L.; Hamilton, L.H.; Lazovich, T.; Harer, J.A.; Ozdemir, O.; Ellingwood, P.M.; McConley, M.W. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In Proceedings of the 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), Orlando, FL, USA, 17–20 December 2018.
19. Ronen, R.; Radu, M.; Feuerstein, C.; Yom-Tov, E.; Ahmadi, M. Microsoft malware classification challenge. *arXiv* **2018**, arXiv:1802.10135.