Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

AutoVAS: An automated vulnerability analysis system with a deep learning approach

Sanghoon Jeon[a,b], *Huy Kang Kim*[b,b,*]

[a] *Samsung Electronics, Suwon, Republic of Korea*
[b] *School of Cybersecurity, Korea University, Seoul, Republic of Korea*

ARTICLE INFO

Article history:
Received 20 October 2020
Revised 23 March 2021
Accepted 19 April 2021
Available online 26 April 2021

Keywords:
Vulnerability detection
Cybersecurity
Data-driven security
Static analysis
Program slicing

ABSTRACT

Owing to the advances in automated hacking and analysis technologies in recent years, numerous software security vulnerabilities have been announced. Software vulnerabilities are increasing rapidly, whereas methods to analyze and cope with them depend on manual analyses, which result in a slow response. In recent years, studies concerning the prediction of vulnerabilities or the detection of patterns of previous vulnerabilities have been conducted by applying deep learning algorithms in an automated vulnerability search based on source code. However, existing methods target only certain security vulnerabilities or make limited use of source code to compile information. Few studies have been conducted on methods that represent source code as an embedding vector. Thus, this study proposes a deep learning-based automated vulnerability analysis system (*AutoVAS*) that effectively represents source code as embedding vectors by using datasets from various projects in the National Vulnerability Database (NVD) and Software Assurance Reference Database (SARD). To evaluate *AutoVAS*, we present and share a dataset for deep learning models. Experimental results show that *AutoVAS* achieves a false negative rate (FNR) of 3.62%, a false positive rate (FPR) of 1.88%, and an F1-score of 96.11%, which represent lower FNR and FPR values than those achieved by other approaches. We further apply *AutoVAS* to nine open-source projects and detect eleven vulnerabilities, most of which are missed by the other approaches we experimented with. Notably, we discovered three zero-day vulnerabilities, two of which were patched after being informed by *AutoVAS*. The other vulnerability received the Common Vulnerabilities and Exposures (CVE) ID after being detected by *AutoVAS*.

1. Introduction

Hidden flaws in software could potentially lead to security vulnerabilities that could allow attackers to compromise systems and applications. With the recent advances in hacking technology, software vulnerabilities have steadily increased in number (NIST, Accessed: Mar 2021e). More than 20,000 security vulnerabilities have been registered in the Common Vulnerabilities and Exposures (CVE) system (NVD, Accessed: Mar 2021), which is a publicly available vulnerability database, in 2019 alone. This demonstrates the rapid increase in software security vulnerabilities. A recent exploit incident (NIST, Accessed: Mar 2021c) showed that these security loopholes can have catastrophic financial and social impacts. These vulnerabilities are often caused by subtle programming errors and can propagate quickly, owing to the spread of open-source software and code reuse. Previously, security experts analyzed software to discover vulnerabili-

* Corresponding author.
E-mail addresses: sh47.jeon@samsung.com (S. Jeon), cenda@korea.ac.kr (H.K. Kim).
https://doi.org/10.1016/j.cose.2021.102308

ties and patched the software themselves. However, analyzing software and searching for vulnerabilities takes considerable time, and the speed at which vulnerabilities can be analyzed depends on the experts skill level, which makes it difficult to achieve a quick response. To solve the technical dependence on experts and reduce the cost of vulnerability detection, techniques (Kim et al., 2017; Li et al., 2018b; Zou et al., 2019) and tools (CheckMarx, Accessed: Mar 2021; Fortify, Accessed: Mar 2021) for automated static vulnerability detection have emerged.

Research on deep learning methods that minimize the intervention of experts and automatically learn a pattern of vulnerabilities has become a new trend in software vulnerability detection (Lin et al., 2020). This can also be justified by the automation of cyber defense, as promoted by initiatives such as the Cyber Grand Challenge (CGC) created by the Defense Advanced Research Projects Agency (DARPA) (DARPA, Accessed: Mar 2021). However, legacy deep learning-based vulnerability detection systems utilize the syntax and semantics of software to improve detection performance, but they have several limitations (Li et al., 2020). First, they have a long-lasting dependence on the context of vulnerable codes. For example, variables defined at the beginning of the program or function are used at the end, or the vulnerability in the source code may call many functions. As a result, a deep-learning algorithm may ignore correlations between contexts when detecting vulnerabilities. Second, these methods have an out-of-vocabulary (OoV) problem when detecting vulnerabilities in new programs in which few identifiers are employed in the source code used for learning. All programmers have unique styles when designating the names of identifiers (variables, functions, etc.) in the source code. As a result, common vocabularies are not sufficient for managing all possible identifiers. This may degrade the vulnerability detection results. Finally, the effect of the deep learning-based vulnerability detection method is highly dependent on the amount and quality of the learning data.

To solve this problem, this study proposes a deep learning-based vulnerability detection framework called an automated vulnerability analysis system (*AutoVAS*). This framework utilizes a compiler-based program slicing method to solve the long context dependence problem and applies various embedding methods and symbolic representation techniques to solve the OoV problem. In addition, various datasets in the National Vulnerability Database (NVD) (NIST, Accessed: Mar 2021e) and Software Assurance Reference Dataset (SARD) (SARD, Accessed: Mar 2021) were used for the source code employed as the learning data to search for various types of security vulnerabilities in the vulnerable datasets, and a learning dataset with 98 vulnerabilities from the Common Weakness Enumeration (CWE) database (NIST, Accessed: Mar 2021b) and 719 vulnerabilities from the CVE was constructed (see Appendix 1). Furthermore, an oversampling method was used to overcome the imbalance problem in the datasets. The contributions of the present study are summarized as follows:

- An optimal method for representing source code as input vectors in a deep learning model was proposed from the viewpoints of program slicing and embedding techniques. *AutoVAS*, which is a deep learning-based vulnerability detection framework, was also proposed, and its effectiveness was verified through experiments.
- The datasets were built based on the NVD and SARD projects and released on GitHub (GitHub, Accessed: Mar 2021b). The source lexing results of the datasets and software were released to the public to be utilized in related studies.
- A full-featured prototype of *AutoVAS* was implemented. The proposed technique achieved an FAR of 1.88%, an FRR of 3.62%, and an F1-score of 96.11%. Eleven vulnerabilities were detected in nine open-source projects. Notably, we discovered three zero-day vulnerabilities, two of which were patched after being informed by *AutoVAS*. The other vulnerability received the CVE ID after being detected by *AutoVAS*.

The remainder of this paper is organized as follows. Section 2 provides the background necessary for the design and implementation of *AutoVAS*. Section 3 presents the detailed design of *AutoVAS*. The evaluation results of the proposed mechanisms are summarized in Section 4. Section 5 discusses some limitations of the current model and possible improvements. Section 6 reviews related work, and finally, Section 7 concludes this paper.

2. Background

First, we describe vulnerability detection in Section 2.1. We then address the program slicing and word embedding method used to make the embedding vector an input of the deep learning model in Sections 2.2 and 2.3. Moreover, in Section 2.4, we introduce terminologies to facilitate the understanding of this research. Finally, in Sections 2.5 and 2.6, we address the threat model and assumptions of this research.

2.1. *Vulnerability detection*

Vulnerability detection is a method for discovering vulnerabilities in software. Existing vulnerability detection techniques employ static and dynamic analysis methods (Brooks, 2017). Static methods, such as data flow analysis (CheckMarx, Accessed: Mar 2021) and theorem proving (Henzinger et al., 2003), analyze source code or executable code without running software. Static analysis methods have the advantages of wide scope and early distribution in the software development phase but have the drawback of a high false-positive rate (FPR). In contrast, dynamic analysis methods, such as fuzzy testing (Böhme et al., 2017) and dynamic symbolic execution (Cadar et al., 2008), execute a program to verify software characteristics and detect vulnerabilities. Dynamic analysis has the advantages of a low FPR and easy distribution, but has the drawback of low coverage, depending on the application scope of the test example. Furthermore, vulnerability detection methods based on existing static and dynamic analyses require experts to define rules or functions to create vulnerability patterns. Thus, research on deep learning methods that minimize the intervention of experts and automatically learn vulnerability patterns has become a new trend in software vulnerability detection.

Vulnerability detection techniques based on deep learning, which are applied to target source code, can be classified into the use of a software engineering index, anomaly detection, and vulnerable code pattern learning (Ghaffarian and Shahriari, 2017). In the early phase, a software engineering index, such as software complexity (Younis et al., 2016), developer activity (Younis et al., 2016), or source code commit (Perl et al., 2015), is used to learn a deep learning model. This approach begins with the assumption that the higher the software complexity, the greater the vulnerability. This method has the advantages of high speed and relatively easy collection of data, but its accuracy and efficiency still need improvement. Anomaly detection and vulnerable code pattern learning detects vulnerabilities by utilizing syntax and semantic information in the source code (Allamanis et al., 2018). Anomaly detection learns a correct programming pattern from normal software code. This technique has the advantages of easy data collection and the ability to discover unknown vulnerabilities, but it has the drawback of a high FPR. In contrast, the vulnerable pattern learning method learns a vulnerable pattern through vulnerable source code samples (Grieco et al., 2016). It has a drawback in that it is highly dependent on dataset quality, but it has a lower FPR and higher accuracy than anomaly detection. Thus, in recent studies, it has been widely used as a learning method for vulnerable patterns. Therefore, in this study, various vulnerability types were detected using both the vulnerability patterns and the corresponding normal patterns as a training dataset.

2.2. *Program slicing*

According to related studies (Li et al., 2018a; 2018b; Russell et al., 2018), from the program graph and instruction sequence perspective, fetching all data from the entire function or execution path causes performance degradation. This is because the difference between the vulnerable and invulnerable code samples is usually small when comparing the overall function (the changes usually include only a few statements), such that most of the functions code may not be related to the vulnerability. As a result, there should be a way to retain important information to detect vulnerabilities while reducing irrelevant noise in the source code.

Program slicing (Weiser, 1984) is a technique for obtaining a set of affected or influenced statements by a programs target point (criterion). The basic idea of program slicing is to remove irrelevant sentences from the source code while preserving the programs meaning. Program slicing has been widely used in many software engineering fields, including software testing, re-engineering, and program analysis (Silva, 2012). To start program slicing, the slicing criterion must be selected. The slicing criterion is denoted by <p, V>, where p is the source codes program point, and V is the points variable. Even for the same source code, if the criterion is different, a completely different slicing result may be obtained. Thus, in this study, to achieve the purpose of vulnerability analysis, the slicing criterion is set based on application programming interfaces (APIs) (GitHub, Accessed: Mar 2021b) with known vulnerabilities in the training dataset. In addition, when the program change history can be checked, the changed point can be designated as the slicing criterion. To perform the program slicing, first,

```
1  int a, b, c, d;
2
3  a = -2;
4  if (a < 0) {
5      b = 4;
6  } else {
7      b = 1;
8  }
9
10 if (b > 3) {
11     d = 3;
12 } else {
13     d = 2;
14 }
15
16 c = d + 1;
17 a += 10;
```

Listing. 1 – Sample code for program slicing.

create a control flow graph (CFG) (Brockschmidt et al., 2018) and a program dependency graph (PDG) (Yamaguchi et al., 2014). All statements are represented as nodes in the PDG, and all dependencies are represented as edges. Next, the program slicing algorithm starts from the slicing criterion and searches the PDG using the breadth-first search (BFS) along with the control and data dependency edge. All traversed statements are added to the program slices.

Program slicing techniques can be classified into static and dynamic slicing according to whether static information or dynamic execution information is used for specific program input data (Zhang, 2019). Because static slicing uses only statically available information, the slicing result is slightly inaccurate and has a path explosion problem. However, static slicing has the advantage of minimizing the intervention of experts in program slicing. On the contrary, dynamic slicing uses data computed on a given input. Because dynamic slicing is performed at runtime, accurate results can be obtained and effectively used for debugging and testing. However, dynamic slicing has a drawback that it requires expert intervention. In other words, static slicing does not consider the execution of the program, but it works on all possible input data. Accordingly, in this study, the static slicing method was used to utilize as much input data as possible.

Program slicing can also be classified into backward slicing and forward slicing based on the tours direction. Backward slicing consists of all the program statements that can affect the slicing criterion and help the developer find code blocks that contain bugs. Forward slicing contains the programs statements affected by the slicing criterion and predicts the code block affected by the modification. Regarding vulnerability detection, it is important to find the cause of the vulnerability and detect the code block affected by the vulnerability; therefore, both forward and backward slicing were used in this study. In the sample code shown in Listing 1 above, if the slicing criterion is <5, b>, the statements included in slicing are lines 3, 4, 5, 10, 11, 13, and 16. In the case of backward slicing,

```
1  void main()
2  {
3      int in_data1, in_data2;
4      int out_data1, out_data2;
5
6      out_data1 = foo(in_data1);
7      out_data2 = foo(in_data2);
8  }
9
10 int foo(int x1)
11 {
12     int x2 = x1;
13     return x2;
14 }
```

Listing. 2 – Sample code for Weiser algorithm's drawback.

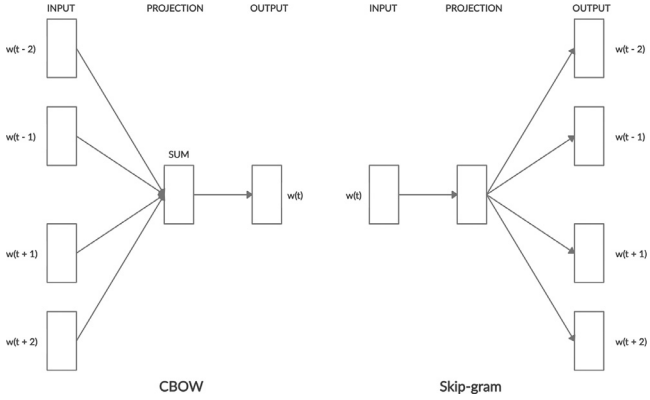

Fig. 1 – CBOW and Skip-gram. CBOW uses surrounding words to predict the central word's meaning, whereas Skip-gram predicts the meaning of surrounding words through the central word.

because the if condition contains b = 4, it has a control dependency, and thus line 4 is included first. The statement in line 3 is added by the data dependency of line 4. In forward slicing, because (a < 0) has data dependency on b = 4, line 10 is added. Then, lines 11 and 13 are added with the control dependency of line 10. Finally, (c = d + 1) of line 16 is added because there is a data dependency on line 11.

The original program slicing method was expressed as a series of data flow analysis problems called the Weiser algorithm (Weiser, 1984). The PDG/SDG (Reps et al., 1995a) algorithm is used based on the reachability problem on the dependence graph, such as the PDG for single-procedural programs and the SDG for multiprocedural programs. The SDG-based algorithm is the most widely used slicing method. In general, to store the dependency between statements/commands, a data structure must be selected along with the CFG, such that the dependency graph includes data flow and control flow information. However, the Weiser algorithms data structure only refers to the data flow set of related variables and statements for the slicing criterion, such that control flow information is not included.

The SDG-based slicing algorithm has advantages over the Weiser algorithm owing to the calling context problem (Horwitz et al., 1990). A well-known disadvantage of the Weiser algorithm is that it cannot solve the calling context problem or the feasible path reachability problem (Reps et al., 1995b). That is, it occurs when the calculation goes down from procedure A to procedure B, and goes up to all procedures that call B, not just A. For example, in Listing 2, an *in_data*1 → *x*1 → *x*2 → *out_data*2 and *in_data*2 → *x*1 → *x*2 → *out_data*1 path can be created, and this execution path cannot be executed. Therefore, incorrect slices are created. Consequently, the Weiser algorithm can generate slices that are less accurate than those generated by the SDG-based algorithm. In addition, the interprocedural finite distributive subset (IFDS)-based algorithm (Naeem et al., 2010) solves several limitations and can perform slicing at a faster rate than the SDG algorithm by using context-sensitive and flow-sensitive data. In this study, both the Weiser and the SDG/IFDS techniques were applied, and their effectiveness was proved through experiments.

2.3. *Source code embedding*

To use source code as an input of a deep learning model, the source code must be represented as a vector (Alon et al., 2019). If the source code is regarded as a sentence, it can be embedded into a vector using several embedding methods in natural language processing (NLP) (Allamanis et al., 2015). Word embedding means that one word constituting the text is converted into a number and expressed in a vector space. This section describes the Word2Vec (Mikolov et al., 2013a), Sent2Vec (Mikolov et al., 2013c), Doc2Vec (Le and Mikolov, 2014), Glove (Pennington et al., 2014), and FastText (Mikolov et al., 2017) methods, which were selected for use in this study among several methods for embedding words. All the aforementioned methods are characterized by preserving the words of co-occurrence.

2.3.1. *Word2Vec, GloVe and FastText*

Word2Vec is a widely used word-embedding method in the field of NLP. Fig. 1 shows two types of Word2Vec: Continuous Bag of Words (CBOW), which matches the central word with the surrounding words, and Skip-gram, a method of predicting the surrounding words with the central word (Mikolov et al., 2013a).

Word2Vec is a method based on the distributional hypothesis. Words located in similar positions have a premise that their intentions will be similar, and based on this, Word2Vec (Skip-gram) aims to maximize Eq. (1).

$$p(o|c) = \frac{exp(U_0^T V_c)}{\sum_{w=1}^{W} exp(U_w^T V_c)} \quad (1)$$

Here, o represents a context word, and c represents the center word. In other words, $p(o|c)$ refers to the the conditional probability that the surrounding word, o, will appear when the central word c is given. Maximizing this expression means matching the surrounding words with the central word effectively. The numerator should be large to maximize the right-hand side of the Eq. (1), and the denomina-

tor should be small. First, to increase the numerators value, the inner product value of the central word, V_c, and the surrounding word, U_o, must be large, which means that the cosine similarity (WIKIPEDIA, Accessed: Mar 2021) must be high. To make the denominator smaller, the inner product values summation of the central word, V_c, and all the learning corpus words must be small. This means that the inner value of the central word and the words that do not appear around it must be reduced (i.e., the cosine similarity should be smaller). Word2Vec embeds a word into a vector by sliding the screen while sliding to the window size in the learning process, looking at the surrounding words for each central word, and updating element values of vectors corresponding to each word continuously. Because learning is performed through window sliding, Word2Vec has a disadvantage in that it is difficult to reflect the co-occurrence information of the entire corpus (Levy et al., 2015).

GloVe (Pennington et al., 2014) is a word-embedding methodology developed by a research team at Stanford University in the United States in 2014. The information GloVe wants to preserve is the words probability of co-occurrence. The inner product between the word vector embedded by GloVe is equivalent to the logarithm of the probability of co-occurrence. The inner product of two-word vector embedding of Word2Vec and GloVe represent the cosine similarity and the probability of co-occurrence, respectively. In other words, GloVe has the advantage of being able to reflect the statistical information of the entire corpus while making it easy to measure the similarity between the embedded word vectors.

Meanwhile, FastText (Mikolov et al., 2017), released by Facebook in 2016, represents and processes the original words as subword vectors, and is otherwise the same as Word2Vec. When words are represented as subwords in FastText, the number of vectors increases, and the ability to cope with an insufficient corpus is improved. In FastText, words are expressed as a bag of character n-grams. For this, first, < and > are added at the beginning and end of a word, respectively. For example, if we use character 3-grams, the word where is expressed as a five 3-letter subword. When expressing actual words, all 3- to 6-grams are used (Pagliardini et al., 2017).

$$\text{"}where\text{"} \rightarrow \text{"} < wh\text{"}, \text{"}whe\text{"}, \text{"}her\text{"}, \text{"}ere\text{"}, \text{"}er > \text{"}$$

In Word2Vec, if the number of occurrences in a word set is high, the embedding vector value is relatively accurate. However, there is a disadvantage in that the accuracy is not high for words with a small number of appearances. However, even for words with a low frequency of appearance, FastText tends to have higher accuracy than Word2Vec because the number of cases that can be referenced increases owing to the nature of embedding with n-grams. Owing to this, FastText has an advantage in a noisy corpus. It would be ideal if there were no typos or misspelled words in each training corpus, but, in reality, unstructured data tend to contain many typos (Bojanowski et al., 2017). Clearly, words with typos are infrequent, and thus they are considered as words with a small number of appearances. In other words, Word2Vec does not properly generate embedding vectors for words with typos, but the performance of FastText is stable, even in this situation.

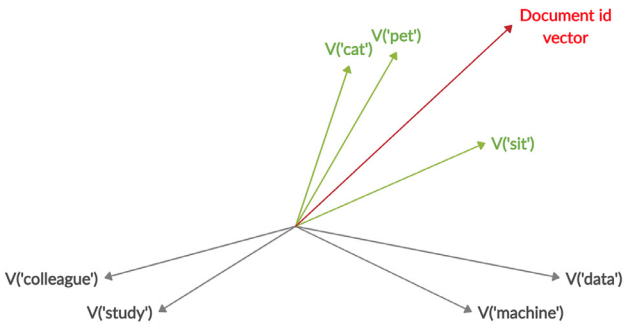

Fig. 2 – Word embedding for Doc2Vec. Even if the words are different, the embedding vectors of each sentence become similar because the words embedding vectors are similar.

2.3.2. *Sent2Vec and Doc2Vec*

Im going to study Math instead of English. and Im going to study English instead of Math. have an identical representation but have completely different meanings. To solve this problem, Sent2Vec (Mikolov et al., 2013c) introduced several techniques that transformed Word2Vecs CBOW to preserve the meaning of sentence-by-sentence learning rather than word-by-word. First, the context window size is not set as a fixed size but is automatically set to cover the entire sentence. In other words, if the sentence is composed of seven words, then the size of the context window is also seven. The context window size matches the number of words in a sentence.

Doc2Vec (Le and Mikolov, 2014) regards the document id as a single word. For example, document id #doc5, which corresponds to the sentence a little dog sit on the table, has positional coordinates in the semantic space. Subsequently, from all the snapshots, the context vector is created by taking the average of the position coordinates of the other words. The remaining operation is the same as that of Word2Vec. That is, Doc2Vec updates the document vector such that the document id and the words appearing in each document (or sentence) approach each other.

Thus, even if the words are different, the embedding vectors of each sentence become similar because the embedding vectors of the words are similar (see Fig. 2). Because the embedding vectors of each word are v(cat) $\simeq$ v(dog) and v(table) $\simeq$ v(board), the embedding vector of each sentence becomes v(a little cat sit on table) ' v(a little dog sit on board).

AutoVAS applied Word2Vec, Doc2Vec, GloVe, and FastText, and their performance was compared to identify the most effective method to embed source code. In the preprocessing stage, after program slicing, user-defined variables and function tokens were generalized. We designed an experiment with the assumption that Sent2Vec or Doc2Vec can be used to achieve an effect similar to that of the generalization method.

2.4. *Terminology*

This study requires an understanding of cybersecurity for vulnerability detection, and deep learning technology for generating and training detection models. We described the background information in the previous subsections. Here, we de-

Table 1 – Terminologies of *AutoVAS*.

Term.	Definitions
Neural network	Like numerous neurons connect to form the brain, a Neural network is a network made by connecting small elements called nodes that correspond to neurons in the brain. In addition, the electrical signals of neurons are created in a way that changes the connection status according to external stimuli, and a Neural network mimics this connection as the weight of the nodes.
Word embedding	Word embedding is a method of representing words as vectors and converts words into the dense representation. The dense representation sets the vector dimensions of all words with the values set by parameters. Also, in this process, it has not only 0 and 1 values, but also a real number. In other words, word embedding is the mapping of words into continuous vectors.
RNN	A recurrent neural network (RNN) is a sequence model that processes inputs and outputs in sequence units. A RNN is known as a model suitable for processing sequential data such as voice and text. Notably, a RNN is the most basic sequence model in deep learning.
Bidirectional RNN	As for text data, backward directional inference, as well as forward directional inference, can produce significant results. However, a standard RNN structure only processes data in the forward direction. Further, because there is no backward direction, future data cannot be used for inference. To overcome this weakness, a bidirectional RNN connects in both directions.
LSTM	RNN's learning ability decreases when the distance between the relevant information and the point where the information is used is long, and the gradient gradually decreases during back-propagation. This is referred to as the vanishing gradient problem. Long short-term memory (LSTM) was designed to overcome this problem. LSTM is a structure in which cell-state is added to the hidden state of the RNN.
GRU	Gated recurrent units (GRU) is a type of RNN that applies a gate mechanism, and GRU reduced the computation of updating the hidden state while maintaining a solution to the long-term dependency problem of LSTM. In other words, GRU has simplified the structure of LSTM, and its performance is similar to LSTM.
Vulnerability	Vulnerability is a weakness used by an attacker to lower the system's information assurance. It is a potential attribute of an asset and is the target of the threat. If there are no vulnerabilities in an asset, the vulnerability can be regarded as a characteristic that establishes a relationship between the asset and the threat since there is no loss even if a threat occurs.
Threat	Threats are events or actions that can adversely affect the inherent vulnerability of assets.
Exploit	As a noun, exploit is a procedure or set of commands or programs designed to perform an attacker's intended actions using a software vulnerability. As a verb, it refers to a hacker's actions to attack assets using a software vulnerability.
CVE	The Common Vulnerabilities and Exposures (CVE) are a dictionary of vulnerabilities discovered in numerous real-world software projects. (*Notation: CVE + year of vulnerability discovery + vulnerability identification number*)
CWE	The Common Weakness Enumeration (CWE) is a set of vulnerability categories. In other words, CWE categorizes significant vulnerabilities and security problems that are generally widely used.

scribe additional terms related to deep learning technology and cybersecurity-related technology, as shown in Table 1.

2.5. *Threat model*

This section describes the threat model of the *AutoVAS* system. Fig. 3 shows a schematic diagram of the *AutoVAS* systems threat model. Software has several vulnerabilities that can be used by a hacker, and exploits are performed when the threat applied to the assets vulnerabilities succeeds. To defend against such hacking attacks, existing safeguards mainly eliminate vulnerabilities through code review or static-rule-based static analysis. However, this method has many limitations in defending against various vulnerabilities. Therefore, in this study, an automated detection system based on deep learning was proposed to detect various vulnerabilities using the CVE and CWE-based datasets. This prevents exploitation by detecting vulnerabilities in the software in advance.

Threats that can be defended through the *AutoVAS* system are affected by the vulnerability type of the dataset used for training. Table 2 categorizes the dataset used in this study based on vulnerability type and shows the corresponding threats. The mapping between vulnerabilities and threats

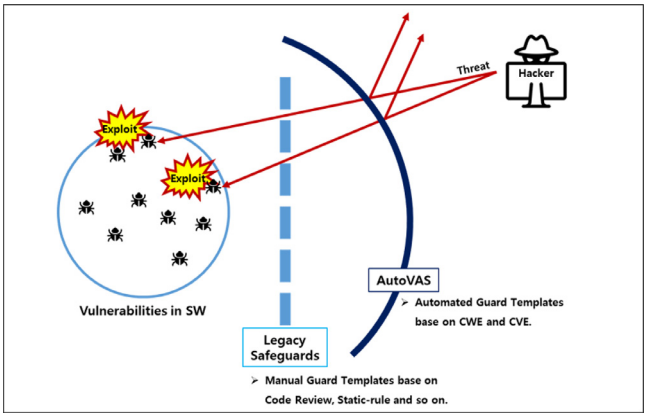

Fig. 3 – Threat Model of *AutoVAS*. When a hacker attacks a vulnerability inherent in software, an exploit occurs. Unlike legacy Safeguards, *AutoVAS* uses an automated template based on CVE and CWE to prevent threats.

is based on the Common Attack Pattern Enumeration and Classification (CAPEC) (MITRE, Accessed: Mar 2021) written by MITRE. However, because various threats can exist for CWE

Table 2 – Mapping between Vulnerability types and Threats.

Vul. Types	Threats
Buffer & Memory	Memory-tainted size allocation Memory-tainted null terminated size Memory-tainted size arguments Memory-unbounded input
Format String & Information Loss	Format wrong call Format taint argument Format buffer overflow
Command Execution	Command wrong call Command tainted argument Command tainted Env.
Improper Error Handling	Using Unicode encoding to bypass logic Lead to a crash or unintended behaviors
Cryptographic Issues	Padding oracle crypto & Cryptanalysis Encryption brute force
Concurrency Issues	Leveraging race condition Leveraging *TOCTOU race condition
Exposed Data & Privileges	Snipping & Interception & Eavesdropping
Etc. (Use of unsafety func.)	Bypass privilege's logic Lead to unintended behaviors

*TOCTOU: Time-of-Check and Time-of-Use

and CVE vulnerabilities, representative threats for each vulnerability type are mapped as targets. In other words, the threat written here is identified to have a higher level of abstraction such that it can represent vulnerabilities with various lower levels of abstraction.

2.6. *Assumptions*

In this study, two assumptions were made to investigate an automated vulnerability detection system based on source code. Previous research (Ghaffarian and Shahriari, 2017) proved that applying the following assumption is effective. Accordingly, this study summarizes this as an assumption and excludes the detailed verification.

First, because the source codes vulnerability is caused by a difference in a part of the entire program and a long-lasting dependency on the program, most of the source code is irrelevant to the vulnerability and must be removed. Accordingly, in this study, meaningful portions of the source code were extracted by applying the program slicing method described in Section 2.2. We use the slicing method to obtain information related to vulnerability, including data and control dependency.

Second, we assumed that program language processing would be similar to NLP. For example, both are composed of tokens and can be represented as a syntax tree. Thus, the deep learning model for word embedding and time-series data processing described in Section 2.3 was applied. The assumptions introduced above are summarized as follows:

- Because most of the program's source code is not related to the vulnerability, the program slicing method is used to extract the source code related to the vulnerability.
- Program language processing is similar to NLP, and thus it uses a deep learning model used in NLP for word embedding and time-series data processing.

3. Main method

The *AutoVAS* process is divided into a training phase to learn the detection model and a detection phase that uses the trained model to detect vulnerable code. As shown in Fig. 4, the training and detection phases use the same embedding process for the input data in the model after preprocessing the source code, except for the activities of model learning and detection. The learning and detecting activities will be explained in Section 4. Thus, this section focuses on the preprocessing and embedding phases. Note that the *AutoVAS* process is explained in the follow order: phase → activity → step → task. That is, in the training phase, there is a preprocessing activity, and within the preprocessing activity, there is a program slicing step, which has a slicing criterion selection task as a subtask.

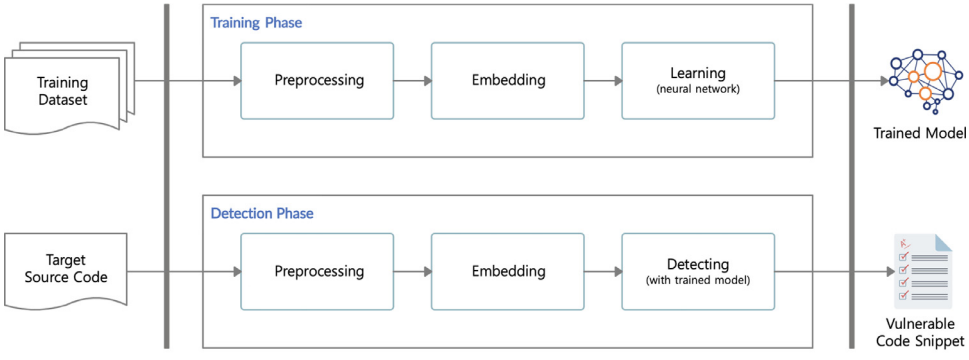

Fig. 4 – Overview of *AutoVAS*. *AutoVAS* consists of a learning phase that trains a model using a dataset and a detection phase that uses the trained model to detect whether the input source code is vulnerable.

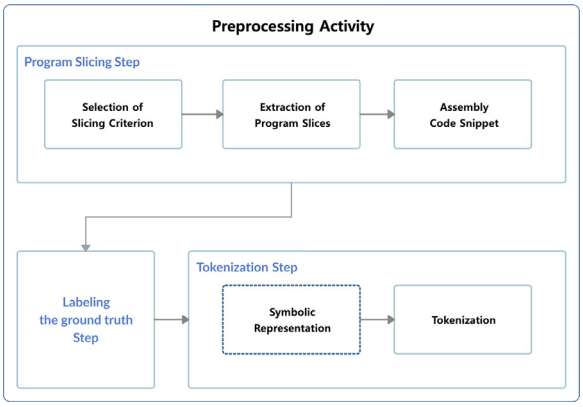

Fig. 5 – Process of preprocessing activity. After synthesizing a code snippet by slicing the source code, tokenization is used to prepare for embedding vector generation.

3.1. *Pre-processing*

A vulnerability in the source code occurs in a series of code statements. Thus, it is not necessary to examine the entire source code to analyze its vulnerability. Instead, a series of source code statements that are related in terms of control flow and data flow around the section where the vulnerability occurs is extracted and analyzed. Accordingly, *AutoVAS* conducts program slicing based on statements where the vulnerability occurred in the source code, or where the vulnerability probability was high. These slices are assembled and marked as a bundle in a series of code statements related to the vulnerability. This bundle is herein referred to as a *code snippet* (Alon et al., 2019). Fig. 5 shows the process flow of the preprocessing activity. The details of the preprocessing activity are explained below.

3.1.1. *Program slicing*

In this study, to express the source code as a vector suitable for input to a neural network, the program is first constructed in the form of a code snippet. A code snippet can be defined as a set of code statements that can define the entire source code from a specific perspective. The difference between the vulnerable and invulnerable code samples is usually small when comparing the overall function (the changes usually include only a few statements), such that most of the functions code may not be related to the vulnerability. As a result, there should be a way to retain important information to detect vulnerabilities while reducing irrelevant noise in the source code. In this study, to compose a code snippet from the viewpoint of vulnerability of the entire source code, parts that are not related to the vulnerability are considered as noise data and excluded. A code snippet is created as a set of code statements related to the vulnerability. When creating a code snippet, it is important to select the key point of the vulnerability occurrence and find related code statements based on the selected point.

In the case of a vulnerability caused by the inappropriate use of an API function, the API function call is the key point. In the case of a vulnerability caused by the inappropriate use of an array, the key point is the array. In addition, certain types of vulnerabilities can have many kinds of key points. For example, buffer error vulnerabilities can have key points of API function calls, arrays, pointers, etc. In particular, this study used the intuition that vulnerabilities mainly occur in parts that use vulnerable APIs and system calls as a key point. Moreover, in a vulnerable code statement, modified portions of patched code were defined as key points. To create a code snippet, a program slicing technique was used. In other words, program slicing was performed considering both data-flow dependency and control-flow dependency to find related code statements based on key points. A detailed description of program slicing is provided in the following subsections. *Selection of slicing criterion*

As described above, to create a code snippet, a key point related to a vulnerability must be selected first. In program slicing, the key point is defined as a slicing criterion. The slicing criterion in *AutoVAS* is a code statement with vulnerability, which is divided into two types. The first criterion is a list of API functions where many vulnerability occurrences have been reported, and the second criterion is a modified code statement that can be verified using the difference between the vulnerability code and patched code.

As is known, a buffer overflow vulnerability may occur in the snprintf and strncpy API functions. Thus, <17, buf1> or <18, buf2> in Listing 3 is selected as the slicing criterion. In addition, if there was no if (argc > 1) statement in line 6 in the previous version and it was patched when releasing, then <7, in str> can also be selected as a slicing criterion. *Extraction of program slices*

As mentioned in Section 2.2, both forward and backward slicing were carried out based on the slicing criterion. Depending on the characteristics of the function or variable, either forward, backward slicing, or both may be needed. For example, assuming that a buffer overflow occurs in the API function *strcpy*, backward slicing is used in the case of the *strcpy* function to extract a code statement that affects the variable causing the overflow in *strcpy*. Alternatively, if a statement that declares a pointer variable is the slicing criterion, it is necessary to find code slices that affect the pointer variable using the forward slicing method. However, performing slicing considering all the above circumstances requires the intervention of experts in the preprocessing activity. Thus, both backward and forward slicing were performed for the criterion, and different assembly strategies were used depending on the slicing type (backward slicing only, forward slicing only, and backward/forward slicing) used in the assembly step to minimize the intervention of experts in this study.

In addition, program slicing can be classified according to the slicing scope and algorithm type. The slicing scope is divided into intraprocedure slicing, which performs slicing in function units, and interprocedure slicing, which considers the calling relationship between functions. We used the Weiser type, including control dependency information, and the SDG-IFDS type, including control dependency and data dependency, as the algorithm types. This study uses llvm-slicing (GitHub, Accessed: Mar 2021d), which is an open-source program slicing tool that is based on the low-level virtual machine (LLVM) compiler. The llvm-slicing tool has the advantage of being able to use compile information such as precise

```c
int main(int argc, void **argv)
{
    char *in_str;
    char *buf1, *buf2;

    if (argc > 1) {
        in_str = argv[1];

        buf1 = malloc(128);
        buf2 = malloc(1024);

        if (!buf1 || !buf) {
            printf("error malloc!\n");
            return -1;
        }

        snprintf(buf1, 512, "<%s>", in_str);
        strncpy(buf2, 512, "<%s>", in_str);

        printf("result: %s %s\n", buf1, buf2);

        free(buf1);
        free(buf2);
    }

    return 0;
}
```

Listing. 3 – Sample code for slicing criterion.

dataflow and control-flow dependency. In addition, the Weiser type and SDG-IFDS type can be selected, and a symbolic program slicing type, a method of improving slicing efficiency, is also provided to be used according to the purpose of slicing. However, the slicing efficiency is outside the scope of this research, and thus we use the Weiser and SDG-IFDS types, focusing on slicing information.

Assembling code snippet

This process converts previously created program slices into a code snippet. Program slices created by the same slicing criterion are first searched, and duplicate slices are removed. Then, the combined slicing types are recorded. That is, if there is only one slicing type (either backward slices or forward slices), the program slices can be a code snippet, or if both backward and forward slices are present, the assembled result of the two slices becomes a code snippet.

When the slicing criterion is selected as <8, sum> in Listing 4, the result of backward slicing is *statement 8* → *statement 5*, and the result of forward slicing is *statement 8* → *statement 11*. Assembling both results as described above, the final result is *statement 5* → *statement 8* → *statement 11*. Even with the different criteria, the same slices can be created as shown above for Listing 4. For example, even if the slicing criterion is selected as <11, sum>, *statement 5* → *statement 8* → *statement 11* is obtained in the same way as for the slicing criterion <8, sum>. Thus, it is necessary to check whether there are duplicate slices, not only for the same criterion but also among all the program slices created from the same source code. This study prioritized the backward slicing result and removed the forward slicing result if the same program slices were obtained from the same source code. Note that the slicing type information, which was recorded when assembling code snippets, was used as important information when splitting or padding the code snippet data to be used later as the input data for the machine learning model. This will be explained in detail when the embedding step is described.

```c
void main()
{
    int i, sum;
    i = 1;
    sum = 0;

    while (i <= 10) {
        sum += 1;
        i++;

        printf("%d\n", sum);
        printf("%d\n", i);
    }
}
```

Listing. 4 – Sample code for Assembling Code Snippet.

3.1.2. *Labeling the ground truth*

Each code snippet is labeled as 1 (vulnerable) or 0 (benign). As shown in Fig. 6, in the NVD dataset used in this study, *VULN* and *PATCHED* keywords were included in the file name. Thus, labeling was conducted based on the file name. The labeling of the other 102 files that did not have a keyword available in the file name was conducted after checking them manually. In the SARD, Good and Bad keywords were used in the function name in the file, and labeling was assigned based on the keyword applied to the function name where the slicing crite-

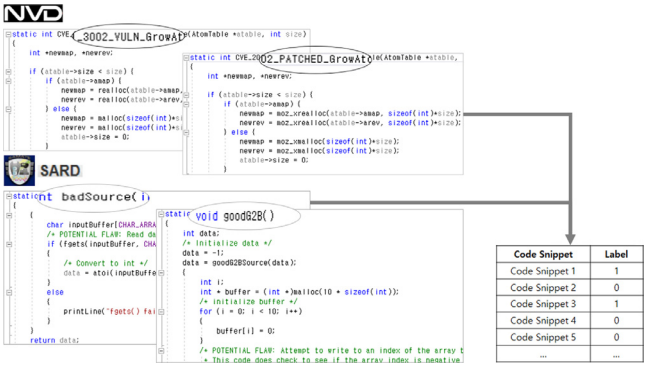

Fig. 6 – Labeling the ground truth. We labeled data based on the function name for the SARD dataset and the file name for the NVD dataset.

```
void code1(int x1)
{
    int x2 = foo(x1);
    char y[3] = {0x00,};

    if (x1 > 10)
        strcpy(y, "pass
    !");
    else
        strcpy(y, "fail
    !");

    printf("%s\n", y);
}

int foo1(int x2)
{
    int y = x2 + 3;
    return y;
}
```

Listing. 5 – Sample 1 for symbolization.

rion was located. In the SARD, there were approximately 430 mixed labeled snippets, and labels were assigned after checking them manually, as was done in the NVD dataset.

3.1.3. *Tokenization*

(Optional) Symbolic Representation

Symbolic representation is a step to change user-defined variables and methods to symbolic names before tokenization. Because user-defined variable and method names may differ even in the source code where they have the same functions, they are given symbolic names considering the scope of naming.

Listings 5 and 6 show a source code example where the left and right sides perform the same function, and a buffer over-

```
void code2(int a1)
{
    int a2 = foo2(a1);
    char b[3]] = {0x00,};

    if (a1 > 10)
        strcpy(b, "pass
    !");
    else
        strcpy(b, "fail
    !");

    printf("%s\n", b);
}

int foo2(int a2)
{
    int b = a2 + 3;
    return b;
}
```

Listing. 6 – Sample 2 for symbolization.

flow vulnerability is detected in the same section. It is necessary to reduce the name variance as much as possible before creating tokens of the above code. Thus, the variables and functions are given symbolic names such as *VAR#* and *FUN#*. Note that even with the same name, variables can play different roles and functions according to the scope of the variables or functions. Thus, different names may be assigned, such as *VAR2* (x2 in code1) and *VAR4* (x2 in foo1), as shown in the above example. This symbolization type is defined as simple symbolization. In addition, the information regarding variable type is important and should be reflected in the symbolic representation. This symbolization type is defined as type-aware symbolization. In addition, the return type and parameter information of functions could be considered, but they were not used in this study.

As mentioned briefly above, the symbolic representation task is intended to reduce the name variance of variables or functions, which may not be effective depending on the embedding method used. Thus, the symbolic representation was set as an optional task in this study. The performance difference observed when using both the mixing symbolic representation method and the embedding method will be reviewed in detail in Section 4. *Tokenization*

Tokenization is the process of splitting source code into token units, for vectorizing code snippets. Fig. 7 shows that a token is the smallest meaningful element in the C/C++ language, such as an operator, separator, keyword, C/C++ standard library, user-defined variable, or user-defined function.

The corpus in *AutoVAS* is entered in the code snippet unit, and a snippet consists of token units. This principle is the same as that of each sentence or paragraph unit being entered into a corpus in NLP, and a sentence or paragraph being composed of words.

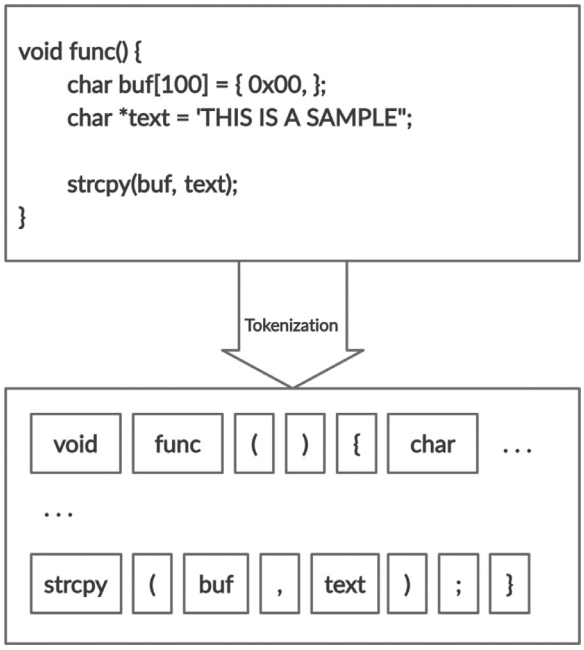

Fig. 7 – Example of tokenization. The code snippet generated through program-slicing is divided into token units.

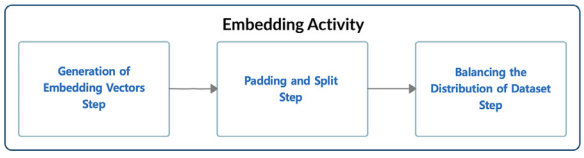

Fig. 8 – Process of embedding activity. A fixed-length embedding vector is created by using the set of code snippets divided by token units as a corpus. Additionally, to create the balancing dataset, the oversampling method is applied to the training dataset.

3.2. *Embedding*

Vector representation should be prepared and used as input data for a deep learning model. As described in Section 2.3, studies on vectorizing natural language in NLP, such as the Word2Vec technique, have been conducted for a long time. The process of vectorizing a corpus consisting of code snippets is explained below. Fig. 8 shows the process followed in the embedding activity. The details of the embedding process are explained below.

3.2.1. *Generation of embedding vectors*

AutoVAS created embedding vectors by applying various embedding methods using the corpus generated in the preprocessing activity. Because each embedding method was explained in the background section, detailed explanations are omitted here. Instead, this section focuses on the process of creating embedding vectors in *AutoVAS*.

For Word2Vec, GloVe, and FastText, tokenized corpuses should be used as input, and for Doc2Vec and Sent2Vec, the identifier information, which is delimited by the sentence or paragraph unit ID, should be additionally entered with the tokenized input.

Once the above embedding method is performed, an n-dimensional vector value is created for each token, and the tokens that create the code snippet are replaced with the vector values to create a vectorized code snippet. Here, embedding methods other than FastText that employ character n-grams will have the OoV problem. In such cases, the problem may be solved by reducing the min count during learning for embedding. However, this study employed the replacement with a zero value.

3.2.2. *Padding and split*

For RNN (recurrent neural network) based LSTM (long short-term memory) (Cho et al., 2014a), or GRU (gated recurrent unit) (Cho et al., 2014b) models, a fixed length of input data is required. This study determined that the time-series (Gamboa, 2017) meaning of source code statements was important, as explained in the assumption in Section 2.6, which was like NLP. Thus, experiments were conducted that targeted RNN-based LSTM and GRU (Su et al., 2016). Thus, code snippets are treated differently depending on the number of tokens. That is, a split is performed for a snippet that has more tokens than the fixed length, whereas padding is performed for a snippet that has fewer tokens than the fixed length. The value of the fixed length was changed using a hyper-parameter, and performance was compared to find the optimal fixed length (refer to the evaluation section). When a split or padding is performed, the information about backward and forward slicing in early program slicing is taken into consideration.

For RNN-based LSTM (Cho et al., 2014a) or GRU (Cho et al., 2014b) models, the length of the input data must be fixed. This study determined that representing source code statements as a time series (Gamboa, 2017) was important, as explained in Section 2.6, which is similar to NLP. Thus, experiments were conducted using targeted RNN-based LSTM and GRU (Su et al., 2016). Code snippets are treated differently depending on the number of tokens. That is, a split is performed for a snippet that has more tokens than the fixed length, whereas padding is performed for a snippet that has fewer tokens than the fixed length. The value of the fixed length was changed using a hyperparameter, and performance was compared to determine the optimal fixed length (refer to Section 4.3.2). When a split or padding is performed, information about backward and forward slicing in the early program slicing is taken into consideration.

Based on a code snippet in which the number of tokens is larger than the fixed length, the front side of the snippet is split because the rear side is likely to contain information about the defect origin in backward slicing, whereas the rear side of the snippet is split in forward slicing. For a snippet consisting of both backward and forward slicing, the split is performed around the slicing criterion. However, based on a code snippet with fewer tokens than the fixed length, padding is performed from the front side in backward slicing because important information is found on the rear side. Padding is performed from the rear side in forward slicing. Similarly, for a snippet consisting of both backward and forward slicing,

padding is performed on both sides around the slicing criterion.

3.2.3. *(Optional) balancing the distribution of dataset*

The preparation of the input data for the deep learning model was completed using the above process. However, the labeled dataset revealed that the vulnerable dataset (Label: 1) was smaller than the benign (Label: 0) dataset. Thus, this study employed the synthetic minority oversampling technique, which is typically used for balancing imbalanced datasets. The performances of the oversampling and undersampling techniques were compared, and oversampling was chosen because of its superior performance. Three algorithms, SMOTE (Chawla et al., 2002), Modified-SMOTE (Hu et al., 2009), and SMOTEENN (Li et al., 2013), which use synthetic approaches that are most frequently used among oversampling techniques, were compared, and SMOTEENN was applied in this work as it showed the best performance.

In this study, two important points were considered when applying the balancing technique. First, if oversampling is applied to all datasets, synthetic sample data will include invalid data and evaluation data, and distortion may occur in the final performance. Therefore, after classifying training, validation, and evaluation data, SMOTEENN should be applied only to the training dataset. Second, if the data class ratio is different, the models precision in terms of selecting the dominant class increases, but the classs recall with a small amount of data rapidly decreases. If we use a balancing technique, the recall increases, but the precision decreases. Because the F1-score is calculated as a harmonic average of recall and precision, the larger the difference between precision and recall, the smaller the F1-score value. Therefore, when using the balancing technique to solve the following imbalance problem, it is necessary to select an algorithm that can lower the rate of decrease in precision and increase the rate of recall. In this study, the algorithm that showed the best performance was selected by comparing the performance based on the F1-score.

4. Evaluation

We designed a series of experiments to evaluate the effectiveness of *AutoVAS* by investigating the following research questions:

- **RQ1: What is the optimal embedding method for vulnerability discovery in source code?** In this study, methods such as program slicing, word embedding, symbolization, and padding and split were used to embed the source code. We experimented to determine the combination of methods that provides the optimal performance. (Section 4.3)
- **RQ2: How effective is *AutoVAS*in practice?** To verify the efficiency of *AutoVAS*, we compared and measured the performance of RNN series models (LSTM, GRU, etc.). We also applied *AutoVAS* to nine open-source projects to verify whether the vulnerability could be detected in practice. (Section 4.4, Section 4.6)
- **RQ3: By what extent is *AutoVAS*more effective compared to other vulnerability detection approaches?** To answer this question, we compared *AutoVAS* with other approaches that used a deep learning model. (Section 4.5)

4.1. *Evaluation metrics*

We utilized the widely used evaluation metrics for vulnerability detection systems (Pendleton et al., 2016). A true positive (TP) indicates the number of detected vulnerable codes, and a true negative (TN) indicates the number of undetected non-vulnerable codes. A false positive (FP) indicates the number of detected non-vulnerable codes, and a false negative (FN) indicates the number of undetected vulnerable codes. The detailed metrics are as follows:

- **False Positive Rate (FPR):** The ratio of false-positive codes to the total codes that are not vulnerable, where false-positive codes are those that are not vulnerable but are detected as vulnerable. $FPR = \frac{FP}{FP+TN}$
- **False Negative Rate (FNR):** The ratio of false-negative codes to the total codes that are vulnerable, where false-negative samples are the vulnerable codes that are not detected as vulnerable. $FNR = \frac{FN}{TP+FN}$
- **Precision (P):** The ratio of true-positive codes to the total codes that are detected as vulnerable. $P = \frac{TP}{TP+FP}$
- **Recall (R):** The ratio of true-positive codes to the total number of vulnerable codes. $R = \frac{TP}{TP+FN}$
- **F1-score (F1):** The overall effectiveness that considers both precision and FNR. $F1 = \frac{2*P*R}{P+R}$

For FPR and FNR, the closer to 0, the better the detection performance; for precision, recall, and F1-score, the closer to 1, the better the detection performance.

4.2. *Experimental setup*

We implemented the LSTM, BLSTM, GRU, and BGRU neural network model in Python using Keras with TensorFlow (Abadi et al., 2016). We ran experiments on a machine with an NVDIA GeForce GTX 2070 GPU and an Intel i7-9700K CPU operating at 3.6 GHz.

This study used NVD and SARD, which are widely used vulnerability source code databases. Because the NVD dataset is based on the CVE database, we can obtain various software vulnerabilities and obtain the patched source information. SARD is a database of vulnerability test cases collected to evaluate code analysis tools. The SARD Juliet Test Suite used in this study comprises small synthetic C/C++ test case samples. The SARD Juliet Test Case has various categories of vulnerable source code and patched source code for it. This test case example is generally simpler than the source code in an actual software project. It may be different from the actual vulnerable code type, but has the advantage of expressing the general source code type according to the type of vulnerability.

As described in Table 3, the *AutoVAS* dataset collected 368 open-source programs (2,033 files) from NVD and 10,655 programs (10,655 files) from SARD. The NVD consists of one or more files containing vulnerable code or patched code, and the SARD consists of both the vulnerable code and the patched code as a test case in one file.

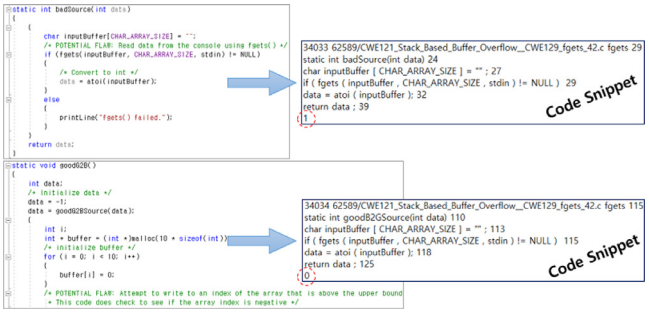

Fig. 9 – Example of code snippets. After forward/backward slicing based on the slicing criteria, the result is assembled to create a code snippet.

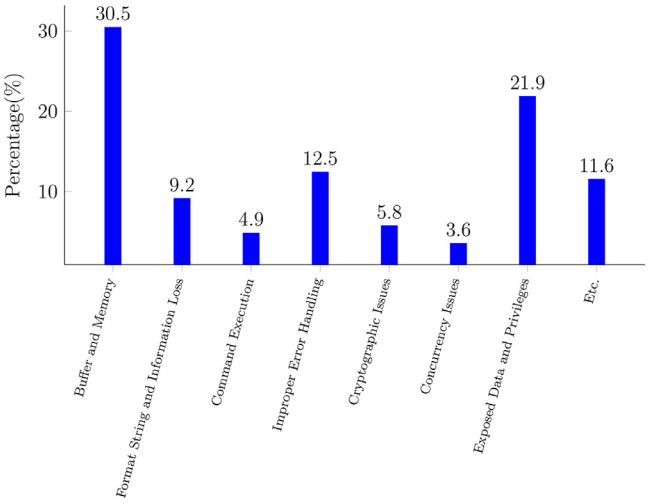

Fig. 10 – Vulnerability types in the dataset. The vulnerabilities included in the NVD and SARD datasets are classified into eight types, indicating the composition ratios. The buffer and memory types account for the largest proportion (30.5%).

We created a total of 145,747 code snippets in the case of intraprocedural slicing and 142,099 code snippets in the case of interprocedural slicing. In NVD dataset used in this study, *VULN* and *PATCHED* keywords were included in the file name. Thus, labeling was conducted based on the file name. The labeling of the other 102 files that did not have a keyword available in the file name was conducted after checking them manually. In SARD, *Good* and *Bad* keywords were used in the function name in the file, and labeling was assigned based on the keyword applied to the function name where the slicing criterion was located. There were approximately 430 mixed labeled snippets in SARD, and labels were assigned after checking them manually, as was done in the NVD Dataset. We created a set of code snippets, as shown in Fig. 9.

We randomly selected 80% of the dataset for training and the remaining 20% for evaluation. Among the training data, 20% was randomly selected for validation. The ratio of the dataset is summarized as follows:

$$training : validation : evaluation = 64\% : 16\% : 20\%$$

The hyperparameters we focused on tuning have a greater impact on the results, according to the deep learning community. The final hyperparameters values that we tuned are listed in Table 4. For other hyperparameters, we chose their default values. The details of the hyperparameters are explained in Section 4.4.

4.2.1. *Vulnerability type of dataset*

The type of vulnerability detectable in the *AutoVAS* system depends on the type of dataset used for training. Therefore, by analyzing the composition status of each vulnerability type in the dataset used in this study, it is possible to check the vulnerability types that can be detected in the *AutoVAS* system.

The SARD data contain a total of 98 CWEs, and the NVD data contain a total of 719 CVEs. Table 14 in Appendix 2 shows the categorized vulnerability of this dataset based on CWE. It was divided into eight categories, and the vulnerability related to coding errors such as the use of unsafe functions was classified as Etc. For a more detailed analysis of vulnerability types, Table 15 in Appendix 3 shows 61 types, including the dataset used in this study among 81 C language vulnerability types (NIST, Accessed: Mar 2021d) defined by NIST. In addition, NIST publishes the Top 40 Most Dangerous Software Weaknesses (NIST, Accessed: Mar 2021a), considering the frequency of occurrence (registration) of vulnerabilities and the degree of impact. Among them nine, indicated in blue in Table 15, are types of vulnerabilities in the C language. This dataset includes all nine types as in Table 15. Therefore, the dataset used in this study covers approximately 76.55% of the vulnerability types of the C language, but it includes all the frequently occurring vulnerabilities, and thus it can be expected that various types of vulnerabilities can be detected.

Fig. 10 shows the composition ratio of the vulnerability types in the dataset used in this study. Buffer and memory-related vulnerability types accounted for the largest with 30.5%, and the exposed data and privileges types accounted for the second largest with 21.9%.

Table 3 – Overview of *AutoVAS* Dataset.

Dataset	Target Files	Code Snippets	Vulnerable	Not Vulnerable
SARD	10,655 files	76,667 (79,583)	30,097 (31,252)	46,570 (48,331)
NVD	2,033 files	65,432 (66,164)	14,194 (14,349)	51,238 (51,815)
Total	12,688 files	142,099 (145,747)	44,291 (45,601)	97,808 (100,146)

Table 4 – Hyper-parameters of neural network models.

Hyper-parameter	LSTM	BLSTM	GRU	BGRU
Input dimension	(200, 80)	(200, 80)	(300, 80)	(200, 80)
Batch Size	256	256	256	256
Epochs	200	200	200	200
Layers	1	3	1	3
Hidden dim	200	300	200	200
Dropout	0.46	0.38	0.42	0.42
Loss function	cross-entropy	cross-entropy	cross-entropy	cross-entropy
Optimizer	Adam	Adamax	Adamax	Adamax
Monitor (Early Stop/Patience)	val_loss / 10	val_loss / 10	val_loss / 10	val_loss / 10

4.3. *Comparison of embedding methods*

This study used program slicing, word embedding, symbolization, and padding and split to convert source code into an embedding vector. We experimented to determine the combination of methods that provides the optimal performance. Each method was treated as an independent variable, and for a fair comparison, the same default settings were applied for each experiment.

4.3.1. *Program slicing*

Program slicing can be classified according to the slicing scope and algorithm type. The slicing scope is divided into intraprocedure slicing, which performs slicing in function units, and interprocedure slicing, which considers the calling relationship between functions. We used the Weiser type, including data dependency information, and SDG-IFDS type, including control dependency and data dependency, as the algorithm types. A comparative analysis of the combination of scope and type is shown in Table 5. The default settings used in this experiment were set to embedding (Word2Vec), symbolization (type-aware), padding and split (combine), snippet size (80), and LSTM.

According to Table 5, the best performance was achieved when using interprocedure and SGD-IFDS, which means that using a code snippet containing a large amount of meaningful information can provide good performance. Thus, it is expected that better performance can be provided if a code snippet, including additional information, is generated by applying static analysis techniques such as static taint analysis in the future.

In addition, we can see that the precision is improved when the slicing range is increased (inter), and the recall is improved when the amount of information is increased (control + data dependency). Accordingly, when designing a vulnerability detection system, it is necessary to extend the slicing scope if it is important to reduce false positives. If it is important to reduce false negatives, it is necessary to use a method that can include a large amount of meaningful information.

4.3.2. *Word embedding*

This study conducted comparative experiments by applying Word2Vec, Doc2Vec, FastText, and GloVe, described in Section 2.3, as the word embedding algorithm. In addition, in the case of Doc2Vec, we assumed that the symbolization effect could be expected because it contains information on the meaning of words in the entire document rather than the meaning between words. To test this assumption, the experiment was performed by applying none, that is, no symbolization, and simple symbolization.

Table 6 shows the performance comparison of the word-embedding algorithm. The default setting is interprocedure + SDG-IFDS slicing, simple symbolization, postpadding, snippet size (80), and LSTM.

From Table 6, we can infer that FastText provides the best performance because of subword learning, which is characteristic of learning with a corpus composed of limited tokens. Doc2Vec is generally known to show good performance in a large corpus, that is, performance improves with the increase in the dataset size. Accordingly, when designing a vulnerability detection system, we recommend applying FastText if dealing with a limited corpus, significant noise, or exception tokens. When a sufficiently large corpus can be secured with an environment with low noise, applying Doc2Vec will provide high performance.

To check the symbolization effectiveness of Doc2Vec, we experimented by changing simple symbolization to none under the same settings as the previous experiment. The experimental results are shown in Fig. 11, and the default settings used in this experiment were set to interprocedure + SDG-IFDS slicing, no symbolization, postpadding, snippet size (80), and LSTM.

Table 5 – Experiment result for program slicing.

Slicing	FPR	FNR	Precision	Recall	F1-Score
Intra + Weiser	5.33%	8.87%	88.11%	91.13%	89.59%
Intra + SGD-IFDS	5.10%	7.19%	88.56%	92.81%	90.63%
Inter + Weiser	4.20%	8.56%	90.68%	91.44%	91.06%
Inter + SGD-IFDS	3.73%	6.61%	91.69%	93.39%	92.53%

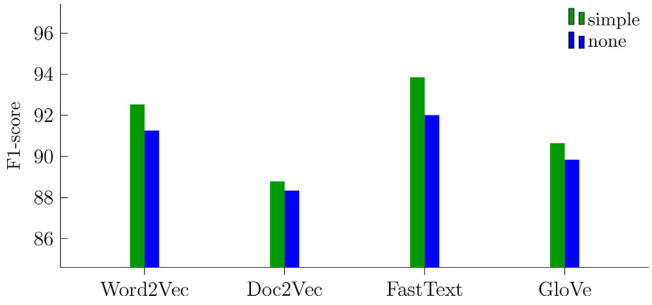

Fig. 11 – Comparison of the symbolization effect of Doc2Vec. The figure shows the comparison performance according to whether the symbolic representation is applied to each word embedding algorithm. It implies that the effects of Doc2Vec are similar to those of symbolization, and the impact will be even more significant if the size of the dataset or corpus increases.

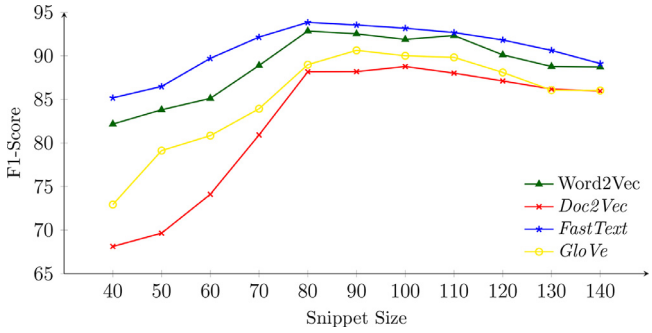

Fig. 12 – Experimental results depending on snippet size. The figure shows the performance change according to the snippet size for each embedding algorithm. The FastText algorithm with a snippet size of 80 achieves the best performance.

Referring to Fig. 11, even when applying no symbolization, FastText still provides the highest performance, but the difference in performance appears smaller than that in Table 6. We can infer that the effects of Doc2Vec are similar to those of symbolization, and the effect will be even greater if the size of the dataset or corpus increases. Accordingly, when designing a vulnerability detection system, it is necessary to select the word embedding algorithm in consideration of the symbolization level and corpus size.

As shown in Fig. 12, to compare the word embedding algorithms performance according to the snippet size, we experimented by increasing the snippet size by 10 units from 40 to 140 for each algorithm.

In Fig. 12, we can see that the optimal snippet size is not significantly affected by the word embedding algorithm, and all algorithms show the optimal performance at a snippet size 80, except for GloVe. Accordingly, the optimal word embedding algorithm should be selected according to corpus characteristics rather than the snippet size.

4.3.3. *Symbolization*

To check the symbolization effectiveness, we experimented by dividing the symbolization levels into none (without symbolization), simple symbolization that identifies variables and functions, and type-aware symbolization that considers the types of variables. The experimental results are shown in Table 7, and the default settings used in this experiment were set to interprocedure + SDG-IFDS slicing, Word2Vec, postpadding, snippet size (80), and LSTM.

According to Table 7, symbolization provides good performance because it helps obtain more meaningful information during embedding. In addition, although there is a slight difference in performance between type-aware and simple symbolization, as with other results, the performance improvement is expected to be greater as the size of the dataset increases.

4.3.4. *Padding and split*

RNN series deep learning models require a fixed input size, and thus padding or splitting input data is required, depending on the snippets length. To check the effectiveness of padding and split, we experimented by dividing them into postpadding and split types, and combine-padding and split types that were applied selectively according to the forward/backward slicing information. The experimental results are shown in Table 8, and the default settings used in this experiment were set to interprocedure + SDG-IFDS slicing, Word2Vec, simple symbolization, snippet size (80), and LSTM.

According to Table 8, the performance gap by the padding and split method is not large. Thus, we performed additional experiments to investigate the effect of changing the snippet size, and the results are shown in Fig. 13 below.

Summarizing the results of Tables 8 and Figure 13, we can see that the performance gap by the padding and split method is not large for the optimal snippet size, but when the snippet size increases, the effect of the combination type is relatively large. It is inferred that the optimal snippet size covers the effect of padding and splitting.

4.3.5. *Experimental summary for optimal embedding methods*

Summarizing the above experimental results, the interprocedure + SGD-IFDS-based slicing method, which can in-

Table 6 – Experiment result for word embedding.

Word Embedding	FPR	FNR	Precision	Recall	F1-Score
Word2Vec	3.73%	6.61%	91.69%	93.39%	92.53%
Doc2Vec	5.23%	10.84%	88.40%	89.16%	88.78%
FastText	3.14%	5.31%	93.02%	94.69%	93.85%
GloVe	4.92%	7.61%	88.95%	92.39%	90.64%

Table 7 – Experiment result for symbolization.

Symbolization	FPR	FNR	Precision	Recall	F1-Score
None	6.21%	10.04%	86.02%	89.96%	87.95%
Simple	3.73%	6.61%	91.69%	93.39%	92.53%
Type-aware	3.44%	6.09%	92.35%	93.91%	93.13%

Table 8 – Experiment result for padding and split.

Padding and Split	FPR	FNR	Precision	Recall	F1-Score
Post	3.73%	6.61%	91.69%	93.39%	92.53%
Combine	3.75%	6.59%	91.66%	93.41%	92.53%

Table 9 – Summary of optimal embedding method.

	Program Slicing	Word Embedding	Symbolization	Padding & Split	Snippet Size
Optimal Method	Inter + SGD-IFDS	FastText	Type-aware	Combine or Post	80 or 90

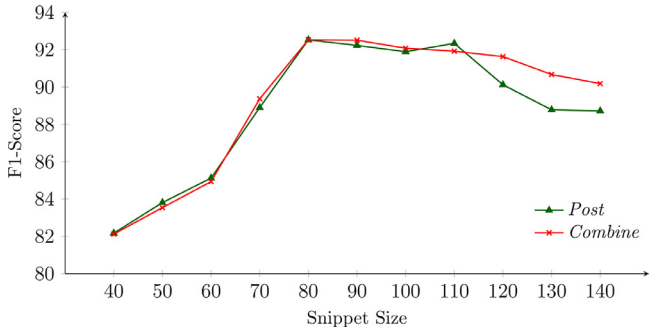

Fig. 13 – Experimental results for the padding and split method depending on snippet size. The figure compares and shows the change in performance according to snippet size for each the padding and split method. It shows that the padding and split method's performance gap is not large for the optimal snippet size, but when the snippet size increases, the effect of the combination type is relatively large.

clude the most meaningful information from the viewpoint of scope and type, performed the best in program slicing. FastText, which is effective in processing none-tokens, shows the best performance among the word embedding methods. In the case of symbolization, type-awareness showed better performance than none and simple symbolization. Finally, there was no significant difference in padding and split, and the snippet sizes of 80 or 90 showed the best performance, regardless of the type of embedding method. Table 9 shows the optimal embedding method.

4.4. *Effectiveness of the* AutoVAS

To check the efficiency of *AutoVAS*, the previously identified optimal embedding method was applied, and then *AutoVAS* was applied to the deep learning model of the RNN series to measure the performance. The tuned hyperparameters described in Section 4.1 were determined while repeating the experiment.

The input dimension refers to the word embeddings output dimension and snippet size, as measured based on FastText, an optimal word embedding algorithm. The best performance was observed with a batch size of 256 at 200 epochs, regardless of the neural network model. The number of neural network layers provided the best performance in multilayers (three layers), except for GRU, which provided the best performance in a single layer. The neural networks hidden dimensions provided the best performance when LSTM was 200, BLSTM was 300, GRU was 200, and BGRU was 200, and the best performance of dropout for LSTM was 0.46, BLSTM was 0.38, GRU was 0.42, and BGRU was 0.42. We achieved the best performance when the loss function was set to cross-entropy, the optimizer was set to Adamax, and the monitor was set to patience 10 based on validation loss.

To find the optimal neural network model in the RNN series, we measured the performance of *AutoVAS* by applying it to various models. The hyperparameters of each model are the same as those listed in Table 4, and the performance is as follows.

According to Table 10, the bidirectional model, rather than the unidirectional model, provides superior performance. LSTM and GRU show similar performance. GRU is known to reduce the computation of updating the hidden state while maintaining a solution to the long-term dependency problem of LSTM, and is known to simplify the complicated structure of LSTM while maintaining performance. Accordingly, it is known that GRU has a faster learning speed than LSTM but shows a similar performance, and the experimental results of this study show that the difference in performance between GRU and LSTM is not significant. In addition, the bidirectional model can accommodate detailed information about the statement appearing before and after the statement; it can be inferred to have better performance than the unidirectional model. (This phenomenon might be explained by the fact that bidirectional RNNs can accommodate more information regarding the statements that appear before and after the statement in question.)

Table 10 – Hyper-parameters of deep learning model.

Model	FPR	FNR	Precision	Recall	F1-Score
LSTM	3.14%	5.31%	93.02%	94.69%	93.85%
BLSTM	2.64%	4.80%	94.14%	95.20%	94.67%
GRU	2.67%	4.76%	94.08%	95.24%	94.66%
BGRU	1.88%	3.62%	95.83%	96.38%	96.11%

4.5. *Comparison of other approaches*

To verify the efficiency of *AutoVAS*, we compared *AutoVAS* to existing deep learning-based vulnerability analysis approaches. In a related study (Ghaffarian and Shahriari, 2017), it was proved that the deep learning-based approach is superior to the static-rule-based approach (FlawFinder (FlawFinder, Accessed: Mar 2021), Checkmark (CheckMarx, Accessed: Mar 2021)) and the code-clone-based approach (VUDDY Kim et al., 2017, VulPecker Li et al., 2016); therefore, we did not compare *AutoVAS* with the static-rule-based approach or code-clone-based approach. As shown in Table 14, we verified that *AutoVAS* outperformed VulDeePecker (Li et al., 2018b) and SySeVR (Li et al., 2018a), state-of-the-art systems that use a deep learning-based vulnerability analysis approach.

VulDeePecker and SySeVR use program slicing similar to *AutoVAS*. However, VulDeePecker includes only data dependency information, and SySeVR includes both data and control dependency information. However, there is a difference in that SySeVR reflects limited information regarding compilation. Moreover, *AutoVAS* differs from previous studies in that it conducts additional tests on how to make source codes into the embedding vector.

The source code of VulDeePecker is not published, and thus we implemented a version with reference to the paper (Li et al., 2018b); we used the published source code of SySeVR from GitHub (GitHub, Accessed: Mar 2021a). We also used the VulDeePecker dataset and SySeVR Data Dependency Control Dependency (DDCD) dataset published on GitHub (GitHub, Accessed: Mar 2021a; Accessed: Mar 2021c). We trained each project dataset and, for a fair comparison, evaluated them by randomly extracting the same ratio from each dataset.

According to Table 11, VulDeePecker demonstrated the lowest performance with an F1-Score of 69.97%. This is because VulDeePecker only includes data dependency information. In particular, because VulDeePecker only considers two types of CWE (NIST, Accessed: Mar 2021b), it cannot detect other types of vulnerabilities and thus shows a high FNR (30.95%). Because *AutoVAS* uses the compile-based slicing method in contrast to SySeVR, it contains more accurate and more information, and also shows the highest performance in all indicators by optimizing the embedding method of the source code. In particular, the FNR improved significantly. This is because various vulnerabilities can be detected by improving the generalization level by optimizing the embedding method, such as symbolization and word embedding.

4.6. *AutoVAS in practice*

To demonstrate the usefulness of *AutoVAS* in detecting vulnerabilities in real-world projects. We applied *AutoVAS* to detect vulnerabilities in nine products: c-ares, Thunderbird, Xen, cJSON, mp3gain, bmp2tiff, boringssl, mpc, and expat.

4.6.1. *Known vulnerabilities*

To evaluate the ability to detect known vulnerabilities, we experimented on the old project version that includes known CVEs. As described in Table 12, we detected seven known vulnerabilities, which were already published in CVE. Notably, the four vulnerabilities represented italic in the c-ares, cJSON, mp3gain, and bmp2tiff project are significant. They were detected only by *AutoVAS* as vulnerabilities that were not detected by the VulDeePecker and SySeVR systems.

4.6.2. *Unknown vulnerabilities*

As described in Table 13, we detected four unknown vulnerabilities. Among them, one vulnerability in the boringssl project was not reported in CVE, but the vendors had silently patched it before being detected by *AutoVAS*. The two vulnerabilities in the mpc project were not registered as CVEs but were delivered to the developer as zero-day vulnerabilities and patched after being informed by *AutoVAS*. Notably, another vulnerability in the expat project received a CVE ID (CVE-2019-15903) as a zero-day vulnerability after being detected by *AutoVAS*.

In summary, *AutoVAS* detected eleven vulnerabilities. Among them, seven were already known, and four were unknown. It is noteworthy that only *AutoVAS* detected four known vulnerabilities represented italic, and three of the unknown vulnerabilities were zero-day vulnerabilities. This result implies that *AutoVAS* can help in efficiently detecting vulnerabilities.

Table 11 – Comparative experiment result with other approaches.

System	FPR	FNR	Precision	Recall	F1-Score
VulDeePecker	8.12%	30.95%	70.91%	69.05%	69.97%
SySeVR	2.95%	8.02%	89.27%	91.98%	90.60%
AutoVAS	1.88%	3.62%	95.83%	96.38%	96.11%

Table 12 – Found Known Vulnerabilities from open-source projects.

Project	Vul. Type	CVE	Version or Commit ID
c-ares	*Memory & Buffer (heap buffer overflow)*	*(known) CVE-2016-5180*	*v1.11.0*
Thunderbird	Memory & Buffer (stack buffer overflow)	(known) CVE-2015-4511	v38.0.1
Xen	Memory & Buffer (integer overflow)	(known) CVE-2016-9104	v4.6.0
	Improper Error Handling (infinite loop)	(known) CVE-2016-4453	v4.7.4
cJSON	*Improper Error Handling (NULL dereference)*	*(known) CVE-2019-1010239*	*v1.7.8*
mp3gain	*Format String & Information Loss (segmentation violation)*	*(known) CVE-2017-14406*	*v1.5.2.r2*
bmp2tiff	*Format String & Information Loss (segmentation violation)*	*(known) CVE-2014-9330*	*v3.8.2*

Table 13 – Found Unknown Vulnerabilities from open-source projects.

Project	Vul. Type	CVE	Version
boringssl	Improper Error Handling (heap use after free)	patched before detected by *AutoVAS*	v3945 (894a47df)
mpc	Memory & Buffer(stack buffer overflow)	(zero-day) patched after being informed by *AutoVAS*	v1.8.4 (b31e02e4)
	Memory & Buffer (heap buffer overflow)	(zero-day) patched after being informed by *AutoVAS*	v1.8.4 (b31e02e4)
expat	Memory & Buffer (heap buffer overflow)	(zero-day) CVE-2019-15903 (registered by *AutoVAS*)	v2.2.8

5. Discussion and limitations

AutoVAS is useful and effective for detecting software vulnerabilities; however, there are still avenues for future improvement. This section presents the limitations of the current *AutoVAS* system and their workarounds and discusses enhancements for future work. - **Dealing with several languages and an executable binary:** The current *AutoVAS* only support the C/C++ language. Future research must be conducted to support other languages. Further, the current *AutoVAS* only detect vulnerabilities of software projects with source code. The detection of vulnerabilities in binary without source code is more challenging. - **Vulnerability detection using unsupervised learning:** The announcement of software vulnerabilities is constantly increasing; however, the dataset for vulnerability detection still lacks vulnerable source codes. This paper uses the imbalanced technique to solve this problem, but there is a limit to detecting the various vulnerabilities more accurately. Accordingly, in vulnerability detection with an imbalanced dataset, research on the unsupervised learning approach rather than the conventional supervised learning is required. In other words, if a large amount of source code without vulnerabilities is used as training data and an abnormal code is detected as a vulnerability, it can effectively detect various vulnerabilities, even when the number of vulnerable source codes is small. - **Searching for the defect origin using Interpretable Machine Learning (IML):** There is a trade-off relationship between fine-grained and coarse-grained vulnerability detection. Using fine-grained analysis, it is easy to find the defect origin after detecting the vulnerability because of its low detection range (usually in terms of functions or files). However, fine-grained analysis provides low detection performance because of the low analysis range and limited information. On the contrary, coarse-grained analysis has a wide scope of detection and provides high detection performance. However, it is not easy to find the defect origin after vulnerability detection, owing to the wide range. To solve this problem, IML or eXplainable AI (XAI) (Tjoa and Guan, 2020) can be applied to find an input token that significantly impacts vulnerability detection, which can be used to analyze the origin of the defect. Even if coarse-grained analysis is used, it is expected that the time and effort required to find the defect origin will be reduced. - **Exploitable vulnerability:** The exploitability of detected vulnerability is the main criterion for determining the vulnerabilitys importance and impact. In particular, by executing the exploit prior to software release, the impact of vulnerability can be predicted in advance. However, not all vulnerabilities detected through vulnerability analysis studies are exploitable. Moreover, it is not easy to analyze exploitable vulnerabilities. Additional research on automatic exploit generation (AEG) (Avgerinos et al., 2014) is needed in existing studies to solve this problem. In other words, it is possible to identify exploitable vulnerabilities among vulnerabilities detected in the existing *AutoVAS* through expanded research on AEG, and identify the priority of the detected vulnerabilities through the importance and impact analysis of these exploits. This expansion of research on AEG allows system administrators and software developers to prioritize security efforts by predicting the importance and impact of vulnerabilities in advance.

6. Related work

With increasing concerns about software vulnerabilities, several studies have been conducted on automated vulnerability detection to reduce developers review efforts. In 2015, DARPA, a research agency under the US Department of Defense, held CGC to encourage research on fully automated software vulnerability analysis systems. These systems are fully autonomous and can perform automated vulnerability detection, exploit generation, and software patching without human intervention (Song and Alves-Foss, 2016). In particular, the Mayhem system that won the CGC showed excellent performance by analyzing vulnerabilities using (Cha et al., 2012) AI technology and applying an exploit generation technique. In addition, CGC targeted programs that intentionally restrict the system calls for the competitions, but several researchers of competitor tools (ForAllSecure, Accessed: Mar 2021; GrammaTech, Accessed: Mar 2021) are working on extending their approaches to fully-fledged OSes. However,

unlike CGC, which is based on binary analysis, the source code-based analysis proposed in this study is still a useful method because it helps reach sufficient analysis coverage more quickly.

This section introduces related research to detect software vulnerabilities statically. There are two approaches to static vulnerability detection for source code: code similarity-based and pattern-based approaches. The code similarity-based approach detects vulnerabilities caused by code cloning, a vulnerable code similar to a given vulnerability code. The pattern-based approach detects matching code using a pattern that can be generated manually or automatically. Deep learning-based vulnerability detection is a pattern-based approach, and pattern-based approaches can be classified into two categories: rule-based and deep learning-based.

6.1. *Code similarity-based approach*

Existing approaches, such as VUDDY (Kim et al., 2017) and VulPecker (Li et al., 2016), detect vulnerabilities by identifying similar code clones. Unlike deep learning methods that learn vulnerable patterns that can be generalized, these methods detect only reused code and the same vulnerability in a large amount of code. There are three steps to the code clone-based approach (Jang et al., 2012; Li and Ernst, 2012). The first step is to divide the program into code fragments. The second step is to represent each code fragment abstractly, such as with a token, tree, or graph. The third step is to improve the similarity between code fragments through the abstract representation obtained in the second step.

Compared to the pattern-based approach, the code similarity-based approach has the advantage of detecting the same vulnerability in the target program because it uses a single instance of vulnerable code. The code clone can be classified into Type I (exact-copy: that is, if all are the same except for spaces and comments), Type II (Type I + only the variable name is different), Type III (Type II + some statements are changed/added/deleted), and Type IV (semantically the same code, but with different syntax) (Rattan et al., 2013). However, this method can only detect vulnerabilities of Type I and Type II code clones and certain cases of Type III code clones. To improve the effectiveness of code clone-based vulnerability systems, experts need to define features for different types of vulnerabilities (Li et al., 2016). Even an advanced approach with expert-defined features cannot detect vulnerabilities that are not caused by code duplication. In other words, when the code clone-based approach is used to detect vulnerabilities that are not caused by code cloning, it has a high FNR. To solve the high FNR in a code-similarity-based approach, in this study, after classifying the vulnerable code containing various vulnerability types and the corresponding benign code into a dataset, a deep learning model was trained to detect various vulnerabilities.

6.2. *Pattern based approach*

6.2.1. *Static rule-based methods*

Static rule-based methods are based on vulnerability rules manually defined by experts. Yamaguchi et al. (2012) detected vulnerabilities using API function usage patterns. Yamaguchi et al. (2014) proposed a method to detect vulnerabilities by combining them into a code property graph (CPG) structure by using CFG, PDG, and abstraction syntax tree (AST) (Bian et al., 2018). In addition, many static vulnerability detection tools that provide rules for each vulnerability are included in this category. This category includes simpler methods of generating patterns from source code (e.g., FlawFinder FlawFinder, Accessed: Mar 2021, RATS RATS, Accessed: Mar 2021, ITS4 Viega et al., 2000, and Checkmarx CheckMarx, Accessed: Mar 2021) and more comprehensive methods of generating patterns based on intermediate source code (e.g., Fortify (Fortify, Accessed: Mar 2021) and Coverity (Coverity, Accessed: Mar 2021)). Rule-based methods can detect the location and type of vulnerabilities. However, they cannot accurately distinguish between various vulnerable codes and invulnerable codes, causing high FPR and FNR (Yamaguchi, 2017). To solve the problem of high FPR and FNR in a static rule-based approach, in this study, a feature was created based on the code snippet generated through program slicing. A deep learning model was then trained for this feature to improve the accuracy of vulnerability detection and analyze various vulnerabilities.

6.2.2. *Deep learning-based methods*

The deep learning-based approach concerns vulnerability detection using deep learning models by inputting various source code information. These existing studies are classified into those using the source codes characteristic information or directly using the source code.

As a method of using the source codes characteristic information, previous studies (Dam et al., 2018; Lin et al., 2018) extracted AST nodes such as method invocation, declaration, and control-flow to capture file-level vulnerability patterns. They proposed a method to detect vulnerability by applying it to a deep learning model such as DBN, LSTM, and Bi-LSTM. In addition, research has been conducted to define the source codes characteristics in various ways, including occurrence frequency (Scandariato et al., 2014), imports and function calls (Neuhaus et al., 2007), complexity, code churn, developer activity (Shin et al., 2010), dependency relation (Neuhaus and Zimmermann, 2009), API symbols, subtrees (Yamaguchi et al., 2012), and system calls (Grieco et al., 2016). The research may also be classified into program, package, component, and file levels with respect to granularity. However, this approach relies on experts to manually define functions and has a limitation in that the exact location of the vulnerability cannot be found because the program is operated with coarse-grained granularity. To solve this problem, in this study, program slicing was performed based on vulnerable system calls and APIs, and the generated code snippet was used for vulnerability analysis. Therefore, when a vulnerability is found, it is possible to analyze where the vulnerability has occurred using control-flow and data-flow information.

As a method of using the source code directly, a previous study (Russell et al., 2018) proposed a method to tokenize the entire source code, generated a vector using Word2Vec, and detected the vulnerability after using it as an input to deep learning models such as convolutional neural networks (CNNs) and RNNs. This study is significant because it is the first attempt to apply the source code to a deep learning model

through embedding. However, there is a limit in that it is difficult to achieve high performance because even code that is not related to vulnerability is used as input data. To solve this problem, VulDeePecker (Li et al., 2018b) proposed slicing the source code, extracting only the necessary part, embedding it, and using it as the input data for the deep learning model. Similarly, SySeVR (Li et al., 2018a) proposed a method to detect the slicing criteria, including data and control dependency information. In addition, uVulDeePecker (Zou et al., 2019) proposed a method to analyze various types of vulnerabilities.

Similar to previous studies, the *AutoVAS* method proposed in this study extracts information through program slicing, embeds it, and uses it as input data for a deep learning model. In VulDeePecker, SySeVR, and uVulDeePecker, because interprocedure and intraprocedure slicing are analyzed on a file-level without information regarding compilation, the accuracy is limited. This study attempted to improve the accuracy by using LLVM-slicing (GitHub, Accessed: Mar 2021d), an open-source slicing tool, to solve the problem of not using compile information. In addition, a previous study (Li et al., 2020) sought to improve the above limitations. However, the pre-training process of converting the source code into an embedding vector is different from the current study. In a study that detects vulnerabilities using source code as input data for a deep learning model, the most important research point is identifying the part of the source code that needs to be vectorized and the corresponding method to be used. Accordingly, this study aims to improve the vulnerability detection performance by proposing a new technique, which is different from the related studies, to transform the source code into an embedding vector. The experimental results show that *AutoVAS*, based on the aforementioned differentiation studies, achieves better performance in the areas of precision, recall, and F1-score compared with other systems such as SySeVR and VulDeePecker.

7. Conclusion

This paper presents *AutoVAS*, an automated vulnerability analysis system based on a deep-learning approach, which aims to relieve human intervention and improve other vulnerability detection systems' low performance. To search for various security vulnerabilities, a dataset was constructed using the source codes of various projects of NVD and SARD. Thus, *AutoVAS* was designed to search for security vulnerabilities of 98 types. This study mitigates the impact of OoV and the lack of vulnerability dataset issues. The optimal method for representing source code as input vectors in a deep learning model was proposed from the viewpoints of program slicing and embedding techniques. Various embedding methods such as program slicing, word embedding, symbolization, and split and padding have been proposed to extract meaningful features, and their effectiveness was verified through experiments. The proposed technique achieved an FPR of 1.88%, an FNR of 3.62%, and an F1-score of 96.11%. Eleven vulnerabilities were detected by applying *AutoVAS* to nine open-source projects. Notably, we discovered three zero-day vulnerabilities, two of which were patched after being informed by *AutoVAS*. The other vulnerability received the CVE ID after being detected by *AutoVAS*. The experimental results show that our method represents an obvious improvement over state-of-the-art methods.

This paper created a publicly available dataset and implemented a prototype. Further, we performed systematic experiments to validate the effectiveness of the proposed method. We are happy to share the source code and dataset of *AutoVAS* on GitHub. The implemented version of *AutoVAS* can be utilized to easily integrate related research. Although there are several studies on deep-learning-based vulnerability detection, there are a number of future research directions, including the limitations of this paper, discussed in Section 5. In particular, the development of embedding methods that consider program characteristics with source code or executable binaries is an exciting and challenging research problem.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper

Appendix A. Vulnerability types of NVD and SARD dataset

The 98 types of vulnerabilities (i.e., CWE IDs) used in the NVD dataset of this study are as follows: CWE015, CWE078, CWE090, CWE114, CWE121, CWE122, CWE123, CWE124, CWE126, CWE127, CWE134, CWE176, CWE188, CWE190, CWE191, CWE194, CWE195, CWE196, CWE197, CWE222, CWE223, CWE226, CWE242, CWE244, CWE252, CWE253, CWE256, CWE259, CWE272, CWE284, CWE319, CWE321, CWE325, CWE327, CWE328, CWE338, CWE364, CWE366, CWE367, CWE369, CWE377, CWE390, CWE391, CWE398, CWE400, CWE401, CWE404, CWE415, CWE416, CWE426, CWE427, CWE457, CWE459, CWE464, CWE467, CWE468, CWE469, CWE475, CWE476, CWE478, CWE479, CWE481, CWE484, CWE506, CWE510, CWE511, CWE526, CWE534, CWE535, CWE562, CWE563, CWE571, CWE587, CWE588, CWE590, CWE591, CWE605, CWE606, CWE615, CWE617, CWE620, CWE665, CWE666, CWE667, CWE675, CWE680, CWE681, CWE685, CWE688, CWE690, CWE758, CWE761, CWE773, CWE775, CWE780, CWE785, CWE789, CWE832

The 719 CVE list used in the SARD dataset of this study are as following URL link: https://github.com/kppw99/AutoVAS/tree/master/resource

Appendix B. Vulnerability types of *AutoVAS* dataset

Table 14 shows the categorized vulnerability of this dataset based on CWE. It was divided into eight categories, and the vulnerability related to coding errors such as the use of unsafe functions was classified as Etc.

Table 14 – Vulnerability types of AutoVAS Dataset.

Vul. Types	CWE List
Buffer & Memory	CWE-015, CWE-121, CWE-122, CWE-123, CWE-124, CWE-126, CWE-127, CWE-188, CWE-190, CWE-191, CWE-244, CWE-400, CWE-401, CWE-404, CWE-415, CWE-416, CWE-464, CWE-476, CWE-588, CWE-590, CWE-665, CWE-680, CWE-690, CWE-761, CWE-773, CWE-775, CWE-785, CWE-789
Format String& Information Loss	CWE-134, CWE-194, CWE-195, CWE-196, CWE-197, CWE-222, CWE-223, CWE-226, CWE-681
Command Execution	CWE-078, CWE-090, CWE-114, CWE-685, CWE-688
Improper Error Handling	CWE-176, CWE-252, CWE-253, CWE-369, CWE-377, CWE-390, CWE-391, CWE-457, CWE-467, CWE-468, CWE-469, CWE-475, CWE-606
Cryptographic Issues	CWE-325, CWE-327, CWE-328, CWE-338, CWE-780
Concurrency Issues	CWE-364, CWE -366, CWE-367, CWE-479
Exposed Data & Privileges	CWE-256, CWE-259, CWE-272, CWE-184, CWE-319, CWE-321, CWE-426, CWE-427, CWE-459, CWE-510, CWE-526, CWE-534, CWE-535, CWE-562, CWE-591, CWE-605, CWE-615, CWE-620, CWE-666, CWE-667, CWE-675, CWE-832
Etc. (Use of unsafety func.)	CWE-398, CWE-481, CWE-478, CWE-484, CWE-242, CWE-506, CWE-511, CWE-563, CWE-571, CWE-587, CWE-617, CWE-758

Appendix C. Vulnerability types of C language

Table 15 shows the 81 possible vulnerability types in C languages defined by NIST. As a result of analyzing the SARD and NVD datasets used in this study, vulnerability corresponding to 61 types are included. In addition, it includes all 9 Top Most Weakness Types in C language.

Table 15 – Vulnerability types of C language.

Vulnerability Type	*Data*	*Top*
Compiler Removal of Code to Clear Buffers	X	
Improper Restriction of Operations within the Bounds of a Memory Buffer	O	✓
Classic Buffer Overflow	O	
Stack-based Buffer Overflow	O	
Heap-based Buffer Overflow	O	
Write-what-where Condition	O	
Buffer Underflow	O	
Out-of-bounds Read	O	✓
Buffer Over-read	O	
Buffer Under-read	O	
Wrap-around Error	O	
Improper Validation of Array Index	O	

Table 15 (*continued*)

Vulnerability Type	*Data*	*Top*
Improper Handling of Length Parameter Inconsistency	X	
Incorrect Calculation of Buffer Size	O	
Use of Externally-Controlled Format String	O	
Incorrect Calculation of Multi-Byte String Length	X	
Improper Null Termination	O	
Reliance on Data/Memory Layout	O	
Integer Underflow (Wrap or Wraparound)	O	
Integer Coercion Error	O	
Unexpected Sign Extension	O	
Signed to Unsigned Conversion Error	O	
Unsigned to Signed Conversion Error	O	
Numeric Truncation Error	O	
Use of Inherently Dangerous Function	O	
Creation of chroot Jail Without Changing Working Directory	X	
Heap Inspection	O	
Race Condition	O	✓
Signal Handler Race Condition	O	
Race Condition in Switch	O	
Race Condition within a Thread	O	
Passing Mutable Objects to an Untrusted Method	X	
Returning a Mutable Object to an Untrusted Caller	X	
Missing Release of Memory after Effective Lifetime	O	✓
Double Free	O	✓
Use After Free	O	✓
Use of Uninitialized Variable	O	
Improper Cleanup on Thrown Exception	O	
Duplicate Key in Associative List (Alist)	X	
Deletion of Data Structure Sentinel	O	
Addition of Data Structure Sentinel	O	
Return of Pointer Value Outside of Expected Range	X	
Use of sizeof() on a Pointer Type	O	
Incorrect Pointer Scaling	O	
Use of Pointer Subtraction to Determine Size	O	
Use of Function with Inconsistent Implementations	O	
NULL Pointer Dereference	O	✓
Missing Default Case in Switch Statement	O	
Signal Handler Use of a Non-reentrant Function	O	
Use of Incorrect Operator	O	
Assigning instead of Comparing	O	
Comparing instead of Assigning	X	
Incorrect Block Delimitation	X	
Omitted Break Statement in Switch	O	
Private Data Structure Returned From A Public Method	X	
Public Data Assigned to Private Array-Typed Field	X	
Use of getlogin() in Multithreaded Application	X	
Use of umask() with chmod-style Argument	X	
Return of Stack Variable Address	O	
Assignment of a Fixed Address to a Pointer	O	
Use of Potentially Dangerous Function	O	

(*continued on next page*)

Table 15 (*continued*)

Vulnerability Type	*Data*	*Top*
Function Call With Incorrect Number of Arguments	O	
Function Call With Incorrect Variable or Reference as Argument	O	
Permission Race Condition During Resource Copy	X	
Unchecked Return Value to NULL Pointer Dereference	O	
Incorrect Type Conversion or Cast	O	✓
Compiler Optimization Removal or Modification of Security-critical Code	X	
Mismatched Memory Management Routines	O	
Improper Address Validation in IOCTL with METHOD_NEITHER I/O Control Code	X	
Exposed IOCTL with Insufficient Access Control	X	
Operator Precedence Logic Error	X	
Use of Path Manipulation Function without Maximum-sized Buffer	O	
Out-of-bounds Write	O	✓
Memory Allocation with Excessive Size Value	O	
Buffer Access with Incorrect Length Value	O	
Buffer Access Using Size of Source Buffer	O	
Numeric Range Comparison Without Minimum Check	O	
Type Confusion	O	
Use of Expired File Descriptor	O	
Improper Update of Reference Count	X	
Improperly Controlled Sequential Memory Allocation	O	

CRediT authorship contribution statement

Sanghoon Jeon: Conceptualization, Methodology, Software, Writing - original draft. **Huy Kang Kim:** Writing - review & editing, Supervision.

REFERENCES

Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, et al. TensorFlow: a system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16); 2016. p. 265–83.

Allamanis M, Barr ET, Devanbu P, Sutton C. A survey of machine learning for big code and naturalness. ACM Comput. Surv. (CSUR) 2018;51(4):1–37. doi:10.1145/3212695.

Allamanis M, Tarlow D, Gordon A, Wei Y. Bimodal modelling of source code and natural language. In: International Conference on Machine Learning; 2015. p. 2123–32.

Alon U, Zilberstein M, Levy O, Yahav E. code2vec: learning distributed representations of code. Proc. ACM Program. Lang. 2019;3(POPL):1–29. doi:10.1145/3291636.

Avgerinos T, Cha SK, Rebert A, Schwartz EJ, Woo M, Brumley D. Automatic exploit generation. Commun. ACM 2014;57(2):74–84. doi:10.1145/2560217.2560219.

Bian P, Liang B, Zhang Y, Yang C, Shi W, Cai Y. Detecting bugs by discovering expectations and their violations. IEEE Trans. Softw. Eng. 2018;45(10):984–1001. doi:10.1109/TSE.2018.2816639.

Böhme M, Pham V-T, Roychoudhury A. Coverage-based greybox fuzzing as Markov chain. IEEE Trans. Softw. Eng. 2017;45(5):489–506. doi:10.1145/2976749.2978428.

Bojanowski P, Grave E, Joulin A, Mikolov T. Enriching word vectors with subword information. Trans. Assoc. Comput.Linguist. 2017;5:135–46. doi:10.1162/tacl_a_00051.

Brockschmidt, M., Allamanis, M., Gaunt, A. L., Polozov, O., 2018. Generative code modeling with graphs. arXiv:1805.08490.

Brooks, T. N., 2017. Survey of automated vulnerability detection and exploit generation techniques in cyber reasoning systems. arXiv:1702.06162v4.

Cadar C, Dunbar D, Engler DR, et al. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs., vol. 8; 2008. p. 209–24.

Cha SK, Avgerinos T, Rebert A, Brumley D. Unleashing mayhem on binary code. In: 2012 IEEE Symposium on Security and Privacy. IEEE; 2012. p. 380–94. doi:10.1109/SP.2012.31.

Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP. Smote: synthetic minority over-sampling technique. J. Artif. Intell. Res. 2002;16:321–57.

CheckMarx, Accessed: Mar 2021. Checkmarx. https://www.checkmarx.com/.

Cho, K., Van Merriënboer, B., Bahdanau, D., Bengio, Y., 2014a. On the properties of neural machine translation: encoder-decoder approaches. arXiv:1409.1259.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y., 2014b. Learning phrase representations using RNN encoder-decoder for statistical machine translation. 10.3115/v1/D14-1179

Coverity, Accessed: Mar 2021. Coverity. https://scan.coverity.com/.

Dam HK, Tran T, Pham TTM, Ng SW, Grundy J, Ghose A. Automatic feature learning for predicting vulnerable software components. IEEE Trans. Softw. Eng. 2018. doi:10.1109/TSE.2018.2881961.

DARPA, Accessed: Mar 2021. Cyber Grand Challenge (CGC). http://www.cybergrandchallenge.com.

FlawFinder, Accessed: Mar 2021. Flawfinder. http://www.dwheeler.com/awnder.

ForAllSecure, Accessed: Mar 2021. ForAllSecure: Mayhem ensures your apps are secure in the face of the unexpected.https://forallsecure.com/.

Fortify, Accessed: Mar 2021. Hp fortify. https://www.hpfod.com/.

Gamboa, J. C. B., 2017. Deep learning for time-series analysis. arXiv:1701.01887.

Ghaffarian SM, Shahriari HR. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: a survey. ACM Comput. Surv. (CSUR) 2017;50(4):1–36. doi:10.1145/3092566.

GitHub, Accessed: Mar 2021a. Database and Source Code of VulDeePecker. https://github.com/SySeVR/SySeVR.

GitHub, Accessed: Mar 2021b. Database of AutoVAS. https://github.com/kppw99/autoVAS. 10.5281/zenodo.4478131

GitHub, Accessed: Mar 2021c. Database of VulDeePecker. https://github.com/CGCL-codes/VulDeePecker.

GitHub, Accessed: Mar 2021d. LLVM-Slicing. https://github.com/zhangyz/llvm-slicing.

GrammaTech, Accessed: Mar 2021. DARPA Cyber Grand Challenge - Team TECHX | GrammaTech. https://www.grammatech.com/.

Grieco G, Grinblat GL, 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; 2016. p. 85–96. doi:10.1145/2857705.2857720.

Henzinger TA, Jhala R, Majumdar R, Sutre G. Software verification with blast. In: International SPIN Workshop on Model

Checking of Software. Springer; 2003. p. 235–9. doi:10.1007/3-540-44829-2_17.
Horwitz S, Reps T, Binkley D. Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang.Syst. (TOPLAS) 1990;12(1):26–60. doi:10.1145/77606.77608.
Hu S, Liang Y, Ma L, He Y. MSMOTE: improving classification performance when training data is imbalanced, vol. 2. IEEE; 2009. p. 13–17. doi:10.1109/WCSE.2009.756.
Jang J, Agrawal A, Brumley D. ReDeBug: finding unpatched code clones in entire os distributions. In: 2012 IEEE Symposium on Security and Privacy. IEEE; 2012. p. 48–62. doi:10.1109/SP.2012.13.
Kim S, Woo S, Lee H, Oh H. VUDDY: a scalable approach for vulnerable code clone discovery. In: 2017 IEEE Symposium on Security and Privacy (SP). IEEE; 2017. p. 595–614. doi:10.1109/SP.2017.62.
Le Q, Mikolov T. Distributed representations of sentences and documents. In: International Conference on Machine Learning; 2014. p. 1188–96.
Levy O, Goldberg Y, Dagan I. Improving distributional similarity with lessons learned from word embeddings. Trans. Assoc. Comput. Linguist. 2015;3:211–25. doi:10.1162/tacl_a_00134.
Li H, Zou P, Wang X, Xia R. A new combination sampling method for imbalanced data. In: Proceedings of 2013 Chinese Intelligent Automation Conference. Springer; 2013. p. 547–54. doi:10.1007/978-3-642-38466-0_61.
Li J, Ernst MD. CBCD: cloned buggy code detector. In: 2012 34th International Conference on Software Engineering (ICSE). IEEE; 2012. p. 310–20. doi:10.1109/ICSE.2012.6227183.
Li X, Wang L, Xin Y, Yang Y, Chen Y. Automated vulnerability detection in source code using minimum intermediate representation learning. Appl. Sci. 2020;10(5):1692. doi:10.3390/app10051692.
Li Z, Zou D, Xu S, Jin H, Qi H, Hu J. VulPecker: an automated vulnerability detection system based on code similarity analysis. In: Proceedings of the 32nd Annual Conference on Computer Security Applications; 2016. p. 201–13. doi:10.1145/2991079.2991102.
Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., 2018a. SySeVR: a framework for using deep learning to detect software vulnerabilities. 10.1109/TDSC.2021.3051525
Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y., 2018b. VulDeePecker: a deep learning-based system for vulnerability detection. 10.14722/ndss.2018.23158
Lin G, Wen S, Han Q-L, Zhang J, Xiang Y. Software vulnerability detection using deep neural networks: a survey. Proc. IEEE 2020;108(10):1825–48. doi:10.1109/JPROC.2020.2993293.
Lin G, Zhang J, Luo W, Pan L, Xiang Y, De Vel O, Montague P. Cross-project transfer representation learning for vulnerable function discovery. IEEE Trans. Ind. Inf. 2018;14(7):3289–97. doi:10.1109/TII.2018.2821768.
Mikolov, T., Chen, K., Corrado, G., Dean, J., 2013a. Efficient estimation of word representations in vector space. arXiv:1301.3781.
Mikolov, T., Grave, E., Bojanowski, P., Puhrsch, C., Joulin, A., 2017. Advances in pre-training distributed word representations. arXiv:1712.09405.
Mikolov T, Sutskever I, Chen K, Corrado GS, Dean J. Distributed representations of words and phrases and their compositionality. In: Advances in Neural Information Processing Systems; 2013. p. 3111–19.
MITRE, Accessed: Mar 2021. CAPEC: Common Attack Pattern Enumeration and Classification. https://capec.mitre.org/.
Naeem NA, Lhoták O, Rodriguez J. Practical extensions to the IFDS algorithm. In: International Conference on Compiler Construction. Springer; 2010. p. 124–44. doi:10.1007/978-3-642-11970-5_8.
Neuhaus S, Zimmermann T. The beauty and the beast: vulnerabilities in red hat's packages.. In: USENIX Annual Technical Conference; 2009. p. 1–14.
Neuhaus S, Zimmermann T, Holler C, Zeller A. Predicting vulnerable software components. In: Proceedings of the 14th ACM Conference on Computer and Communications Security; 2007. p. 529–40. doi:10.1145/1315245.1315311.
NIST, Accessed: Mar 2021a. 2020 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html.
NIST, Accessed: Mar 2021b. Common Weakness Enumeration (CWE). http://cwe.mitre.org.
NIST, Accessed: Mar 2021c. CVE-2017-5638 Detail. https://nvd.nist.gov/vuln/detail/CVE-2017-5638.
NIST, Accessed: Mar 2021d. CWE VIEW: Weaknesses in Software Written in C. https://cwe.mitre.org/data/definitions/658.html.
NIST, Accessed: Mar 2021e. National Vulnerability Database. https://nvd.nist.gov/.
NVD, Accessed: Mar 2021. Common Vulnerabilities Exposures (CVE). http://cve.mitre.org.
Pagliardini, M., Gupta, P., Jaggi, M., 2017. Unsupervised learning of sentence embeddings using compositional n-gram features. 10.18653/v1/N18-1049
Pendleton M, Garcia-Lebron R, Cho J-H, Xu S. A survey on systems security metrics. ACM Comput. Surv. (CSUR) 2016;49(4):1–35. doi:10.1145/3005714.
Pennington J, Socher R, Manning CD. GloVe: global vectors for word representation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP); 2014. p. 1532–43. doi:10.3115/v1/D14-1162.
Perl H, Dechand S, Smith M, Arp D, Yamaguchi F, Rieck K, Fahl S, Acar Y. VCCFinder: finding potential vulnerabilities in open-source projects to assist code audits. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security; 2015. p. 426–37. doi:10.1145/2810103.2813604.
RATS, Accessed: Mar 2021. Rats. https://code.google.com/archive/p/rough-auditing-tool-for-security/.
Rattan D, Bhatia R, Singh M. Software clone detection: a systematic review. Inf. Softw. Technol. 2013;55(7):1165–99. doi:10.1016/j.infsof.2013.01.008.
Reps T, Horwitz S, Sagiv M. Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages; 1995. p. 49–61. doi:10.1145/199448.199462.
Reps T, Horwitz S, Sagiv M. Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages; 1995. p. 49–61. doi:10.1145/199448.199462.
Russell R, Kim L, Hamilton L, Lazovich T, Harer J, Ozdemir O, Ellingwood P, McConley M. Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA). IEEE; 2018. p. 757–62. doi:10.1109/ICMLA.2018.00120.
SARD, Accessed: Mar 2021. Software assurance reference dataset. https://samate.nist.gov/SRD/index.php.
Scandariato R, Walden J, Hovsepyan A, Joosen W. Predicting vulnerable software components via text mining. IEEE Trans. Softw. Eng. 2014;40(10):993–1006. doi:10.1109/ISSRE.2014.32.
Shin Y, Meneely A, Williams L, Osborne JA. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Trans. Softw. Eng. 2010;37(6):772–87. TSE.2010.81
Silva J. A vocabulary of program slicing-based techniques. ACM Comput. Surv. (CSUR) 2012;44(3):1–41.

Song J, Alves-Foss J. The DARPA cyber grand challenge: a competitor's perspective, Part 2. IEEE Secur. Privacy 2016;14(1):76–81. doi:10.1109/MSP.2016.14.

Su, J., Tan, Z., Xiong, D., Ji, R., Shi, X., Liu, Y., 2016. Lattice-based recurrent neural network encoders for neural machine translation. arXiv:1609.07730.

Tjoa E, Guan C. A survey on explainable artificial intelligence (XAI): toward medical XAI. IEEE Trans. Neural Netw. Learn.Syst. 2020. doi:10.1109/TNNLS.2020.3027314.

Viega J, Bloch J-T, Kohno Y, McGraw G. ITS4: a static vulnerability scanner for C and C++ code. In: Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00). IEEE; 2000. p. 257–67. doi:10.1109/ACSAC.2000.898880.

Weiser M. Program slicing. IEEE Trans. Softw. Eng. 1984(4):352–7.

WIKIPEDIA, Accessed: Mar 2021. Cosine-similarity. https://en.wikipedia.org/wiki/Cosine_similarity.

Yamaguchi F. Pattern-based methods for vulnerability discovery. It-Inf. Technol. 2017;59(2):101–6. doi:10.1515/itit-2016-0037.

Yamaguchi F, Golde N, Arp D, Rieck K. Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy. IEEE; 2014. p. 590–604. doi:10.1109/SP.2014.44.

Yamaguchi F, Lottmann M, Rieck K. Generalized vulnerability extrapolation using abstract syntax trees. In: Proceedings of the 28th Annual Computer Security Applications Conference; 2012. p. 359–68. doi:10.1145/2420950.2421003.

Younis A, Malaiya Y, Anderson C, Ray I. To fear or not to fear that is the question: code characteristics of a vulnerable function with an existing exploit. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy; 2016. p. 97–104. doi:10.1145/2857705.2857750.

Zhang, Y., 2019. SymPas: symbolic program slicing. arXiv:1903.05333.

Zou D, Wang S, Xu S, Li Z, Jin H. μ VulDeePecker: a deep learning-based system for multiclass vulnerability detection. IEEE Trans. Dependable Secure Comput. 2019. doi:10.1109/TDSC.2019.2942930.