


## Article

# False Alarm Reduction Method for Weakness Static Analysis Using BERT Model

Dinh Huong Nguyen, Aria Seo, Nnubia Pascal Nnamdi and Yunsik Son \* 

Department of Computer Science and Engineering, Dongguk University, Seoul 04620, Republic of Korea; hopekr@dgu.ac.kr (D.H.N.); seoaria@dgu.ac.kr (A.S.); nubee@dgu.ac.kr (N.P.N.)

\* Correspondence: sonbug@dongguk.edu

**Abstract:** In the era of the fourth Industrial Revolution, software has recently been applied in many fields. As the size and complexity of software increase, security attack problems continue to arise owing to potential software defects, resulting in significant social losses. To reduce software defects, a secure software development life cycle (SDLC) should be systematically developed and managed. In particular, a software weakness analyzer that uses a static analysis tool to check software weaknesses at the time of development is a very effective tool for solving software weaknesses. However, because numerous false alarms can be reported even when they are not real weaknesses, programmers and reviewers must review them, resulting in a decrease in the productivity of development. In this study, we present a system that uses the BERT model to determine the reliability of the weakness analysis results generated by the static analysis tool and to reduce false alarms by reclassifying the derived results into a decision tree model. Thus, it is possible to maintain the advantages of static analysis tools and increase productivity by reducing the cost of program development and the review process.

**Keywords:** software weakness; weakness analysis; static analysis; false alarm reduction; BERT



**Citation:** Nguyen, D.H.; Seo, A.; Nnamdi, N.P.; Son, Y. False Alarm Reduction Method for Weakness Static Analysis Using BERT Model. *Appl. Sci.* **2023**, *13*, 3502. <https://doi.org/10.3390/app13063502>

Academic Editors: Howon Kim and Thi-Thu-Huong Le

Received: 29 December 2022

Revised: 4 March 2023

Accepted: 6 March 2023

Published: 9 March 2023



**Copyright:** © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

Recently, software has been widely applied in various fields, such as cloud, enterprise software, and the Internet of Things (IoT), and cyber infrastructure has become the basis of modern society. However, as the size and complexity of software increase, security attack problems continue to arise owing to potential software defects. Approximately 75% of recent attempts at Internet attacks exploit software weaknesses [1]. Because of these software weaknesses attacks, software infringement accidents that lead to financial damage owing to leakage of sensitive and important information are increasing. Therefore, the software development life cycle (SDLC) is becoming very important for analyzing and eliminating the weaknesses of software at the time of development. Almost all related software development fields acknowledge this issue and there are studies on detecting weaknesses, preserving project integrity, and applying new technology to enhance security in all aspects [2–4].

Software weaknesses (namely as weakness) usually occur when security requirements are not defined, design with logical errors is performed, coding rules with technical vulnerabilities are applied, software placement is inappropriate, or proper management or deployment of discovered weaknesses is not performed [5]. Important information processed by the system is exposed as a result of such weaknesses and services are normally rendered impossible. In addition, even if the software itself is secure, new weaknesses may appear during the process of linking it to other software or exchanging data. These software weaknesses accelerate damage to a wide range of targets and have a direct impact on national infrastructure. Therefore, software security should be applied throughout the entire process, including software-driven operating systems, driver software, and end-user applications.

Recently, there has been active research on static analysis techniques that can check the weaknesses of software source code during development. Because each static analysis tool has different types of weaknesses that can be analyzed and analysis performance, it is common to use multiple static analysis tools together. However, when multiple tools are used together, the false detection rate also increases, causing many false alarms.

Meanwhile, research on analyzing weaknesses using deep learning models is gaining traction [6–12]. Through automation, deep learning-based weakness analysis techniques can reduce the burden of manually defining weaknesses in traditional static analysis techniques, expand the scope of weakness detection, and increase accuracy. However, it is challenging to analyze syntax or semantics within a programming language because of the variety of programming languages and source codes written by developers [13].

In this study, we present a false alarm reduction method utilizing BERT and decision tree models for reliability analysis of static analysis results. This technique first analyzes line-level weaknesses using the BERT model, which is based on Transformer architecture and has the advantage of solving syntax or semantics problems. In addition, weaknesses were analyzed using multiple static analysis tools to expand the scope of the analysis. Subsequently, the line-level weakness analysis results and the weakness analysis results generated by multiple static analysis tools were evaluated to determine the reliability of each analysis result. Finally, false alarms can be reduced by reclassifying the derived results using a decision tree model. This can reduce the cost of the program development and review process because it can effectively reduce false alarms while maintaining the advantages of static analysis tools.

## 2. Related Work and Background

### 2.1. Software Weakness

Software weaknesses in open sources reported over the past decade have topped the lists with C/C++ accounting for 50% of the seven languages. C/C++ is a powerful programming language used in many practical applications, and software developers often use it to build operating systems, embedded systems, system components, and games [14]. Typical C/C++ projects include Git [15], OpenSSH [16], PuTTY [17], and FileZilla [18]. The C/C++ language has been in use for a longer time than other languages and contains the most code. The higher the amount of code, the more weaknesses occur, and the reasons for the weaknesses in the C language are its more complex syntax, lack of exception processing, and difficulty managing memory compared to some modern languages. In addition, the use of pointers and global variables in C++ language observation has increased, resulting in easy memory corruption and less flexibility in logic processing. Common software weaknesses in the C and C++ programming languages include buffer overflow, string vulnerability, pointer substitutes, dynamic memory management, and race conditions. Projects related to CWE [19], CAPEC [20], and CVE [21] have emerged to define or evaluate the weaknesses of this software. In addition, the CWE Top 25 [22] and OWASP Top 10 [23], which are the results of the evaluation of the risk and occurrence of software weaknesses, are disclosed using the self-evaluation prevention theory of each project. The CWE Top 25 is a list of 25 software weaknesses that can lead to serious weaknesses in software. The OWASP Top 10 is the top 10 major software weaknesses that occur in the web environment.

Command Weakness Enumeration (CWE) is a system for classifying weaknesses in software and hardware [19]. This is maintained by a community project aimed at understanding software flaws and creating automation tools to identify, correct, and prevent flaws. The CWE classification is hierarchically organized based on universality and specificity. Currently (in December 2022, version 4.9), there are 933 weaknesses in the inventory such as Stack-based Buffer Overflow (CWE-121), Improper Limitation to a Restricted Directory ('Path Traversal') (CWE-22), and Use of Hard-coded Credential (CWE-798), etc.

In this study, software weaknesses are analyzed for the C/C++ programming language, which has a complex syntax or semantics and the largest number of weaknesses among

programming languages. In addition, to standardize the weaknesses that each tool can analyze, it is systematized by referring to the CWE system.

### *2.2. Weakness Static Analysis Technique and Tools*

Static analysis can be implemented even when the program is not executed, as a method of directly handling the source code without executing the program. Therefore, it has the advantage of detecting weaknesses early and using them frequently in the development process. Owing to the recent continuous research on weakness static analysis algorithms, models and tools are becoming increasingly powerful. Current static analysis techniques include lexical analysis, type inference, data flow analysis, symbol execution, theorem proving, and model checking.

Lexical analysis divides a program into several small pieces based on a grammatical structure analysis similar to that of a C compiler and then compares these pieces with a loophole library to determine if there is a loophole. For accurate analysis, the interaction between the syntax, semantics, and subroutines of the program should be considered. Type inference occurs when a compiler deduces the types of variables and functions and determines whether access to variables and functions is in accordance with formal rules. Data flow analysis refers to the collection of semantic information from program codes and uses algebraic methods to determine the definition and use of variables during compilation. Symbolic execution uses symbol values rather than actual data to represent the input of a program and generate an algebraic representation of the input symbol in the execution process. Using the constraint resolution method, symbolic execution can detect the possibility of an error. Theorem provision is based on the semantic analysis of the program and can solve problems in infinite-state systems. The proof of the theorem first transforms the program into a logical formula and then uses rules to prove that the program is a valid theorem. The model-checking process first constructs a type of model for a program, such as a state machine or a directional graph, and then compares it through the model to ensure that the system meets predefined characteristics.

The goal of the weakness analysis tool using static analysis technique is to assist developers to develop software more cautiously by the early detection of suspicious structures, use of unsafe API, or dangerous runtime errors. As software security has become more important in the past few years, the market for weakness static analysis tools has expanded. In particular, the most diverse and numerous tools have been developed for C/C++ and JAVA programming languages. Typical tools for weakness static analysis of C/C++ source code targets are Cppcheck [24], Clang Static Analyzer [25], Flawfinder [26], CodeQL [27], Infer [28], SonarQube [29], PVS-Studio [30] and Frama-C [31].

There are advantages and disadvantages depending on the method employed because each tool has a distinct target or range of software weaknesses to be analyzed. Therefore, using a single tool may reduce the scope or accuracy of the analysis of weaknesses. In this study, we used multiple static analysis tools rather than a single static analysis tool to extend the scope of weaknesses and increase the accuracy of the analysis results.

### *2.3. Deep-Learning-Based Weakness Analysis*

Traditional static analysis methods have limitations in that the weakness analysis patterns must be defined manually. To overcome these limitations, deep learning models may be used. Deep learning is a machine learning method that allows artificial neural networks to train sample data. In general, this is realized by classifying program codes with and without weaknesses into vectors and classifying multiple classes that identify types of weaknesses [8]. Typical models used for preprocessing input data for code scanning include token-based and graph-based models. For example, VulDeePecker [11] utilized symbolic representations for program slices, and Devig [12] used graph embeddings for code attribute graphs (i.e., AST, CFG, and DFG). Junho Jeong et al. [32] have researched a data type inference method based on long short-term memory by improved features for weakness analysis in binary code. SySeVR [10] focused on overcoming the defects in the

original model and obtaining program representations that could accommodate syntax and semantic information associated with weaknesses. They detected weaknesses with a more granular slice-level segmentation than VulDeepcker [11]. Zaharia et al. [6] researched a security scanner able to use automated learning techniques based on machine learning algorithms to recognize patterns of weaknesses in source code. Ziems et al. [9] proposed the use of Transformers to address the problem of weakness detection. They used a traditional BERT architecture pretrained in English text to identify flaws in computer code. Moreover, a multi-class model was constructed under the assumption that each sample belonged to a unique label (CWE).

Although deep learning has made significant progress in automation and accuracy in the diversity of weaknesses, complexity, characteristics of the programming language used, and conventional programming methods, some problems still exist. In addition, various deep learning models have been developed; however, there are performance differences based on the architecture or algorithm used. Thus, a deep learning model that can handle the source code's syntax or semantics is required.

#### 2.4. Bidirectional Encoder Representation with Transformers (BERT)

Transformer was first proposed in 2017 to solve sequence-to-sequence tasks while easily processing long-range dependencies [33]. It outperformed alternative neural models in natural language understanding and generation task performance, becoming a representative architecture of natural language processing (NLP). The construction of this architecture has an encoder-decoder architecture and similar internal structures, including attention, feedforward, and normalization layers. The tokenizer parses the input sequence into a token. Each token is then transformed into a vector using word embedding, and the location information is added to the embedding. It also parallelizes many calculations by allowing the tokens to flow independently through the stack.

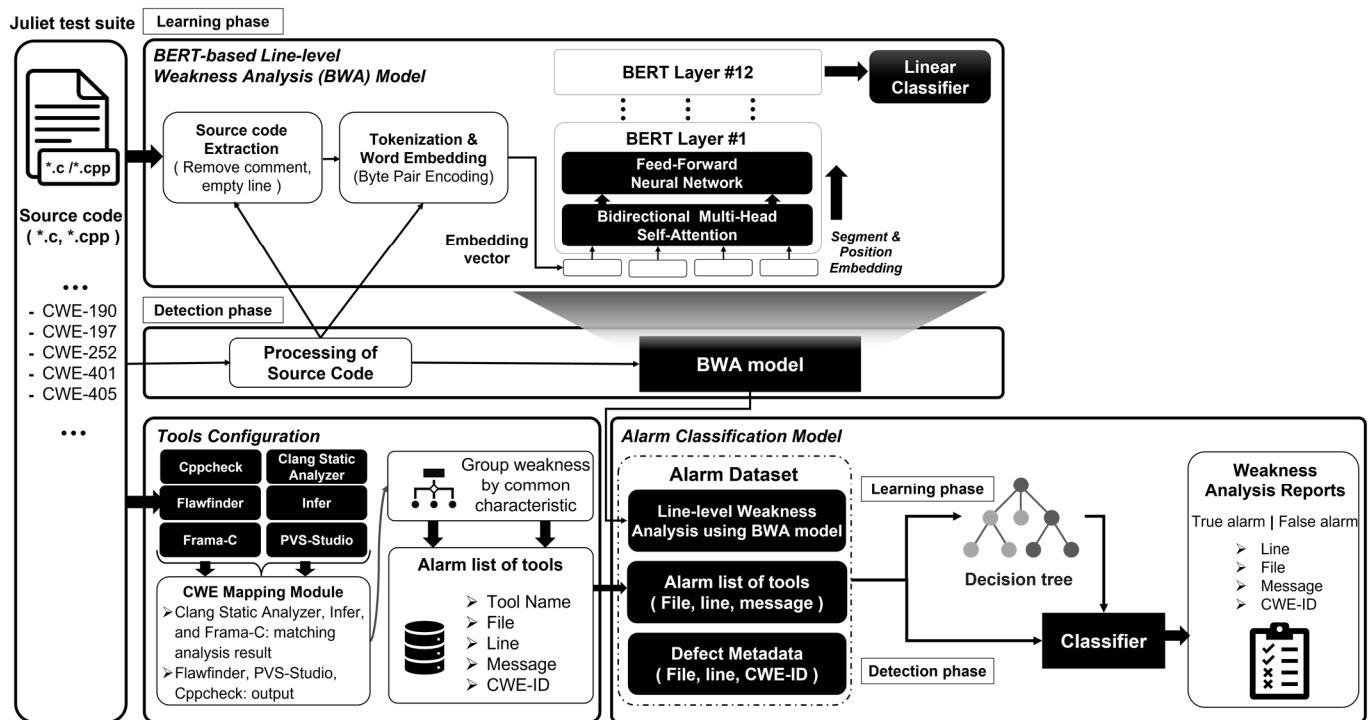
BERT, which stands for Bidirectional Encoder Representation with Transformers, is a bidirectional transformer pretrained using a combination of masked language modeling objective and next sentence prediction on a large corpus of English text [34]. That is, the learning information is from left to right and right to left. It used only the encoder blocks from the original Transformer architecture. In the BERT model, the self-attention mechanism [33] is a key component of the Transformer, and is a process that allows better encoding of words at the current target location by looking at and taking hints at words at different locations in the sentence while processing each word in the entered sentence. This is accomplished by extracting the contextual relationship between the three components, *queries*, *keys*, and *values*. Typical areas to which BERT is applied include single-sentence classification, two-sentence relationship classification, or two-sentence similarity, relationship, sentence token classification, and machine reading answer classification. A representative study applying the BERT model in the field of programming languages is a multi-programming language model pretrained for programming language and natural language pairs, such as CodeBERT [35], GraphCodeBERT [36], and CodeT5 [37].

In this study, the BERT model analyzes weaknesses within C/C++ source codes, which have complex syntax and semantics, based on the characteristics of excellent processing of semantics of natural language text. Using this model, it is not only possible to analyze a specific line with weaknesses but also to obtain a score on the relationship between the weakness lines analyzed for each line based on the line by using the self-attention mechanism where weaknesses occur.

### 3. False Alarm Reduction Method for Reliability Analysis of Static Analysis Results

The system presented in this study comprises three models: a BERT-based Line-level weakness analysis model, a configuration for integrating analysis results of multiple static analysis tools model, and an alarm classification model. Figure 1 shows the structural diagram of the system presented in this study. The BERT-based Line-level Weakness Analysis (BWA) model tokenizes and embeds the input C/C++ source code and then trains

and analyzes the weakness pattern. The configuration for integrating analysis results of multiple static analysis tools uses several static analysis tools to analyze the weaknesses within the source code. The Alarm Classification Model (ACM) is a model for classifying alarms (true and false alarms) based on the decision tree model, which enters the evaluation results of the analysis of multiple static analysis tools and line-level weaknesses analysis through the BWA model. As a result, the ACM reclassifies whether each static analysis result is a true or a false alarm.



**Figure 1.** Architecture of false alarm reduction method on weakness static analysis using BERT model: (\*) represents files with c or cpp extensions.

The dataset used in this study is the Juliet test suite [38], developed by the Center for Assured Software of the U.S. American National Security Agency (NSA), and was first published in December 2010. It consists of relatively short codes characterized by control flow, data flow, or usage data structure and type. The Juliet test suite used is the 1.3 version and it contains a total of 118 weakness classes. C/C++ officially contains 64,099 test cases, including 53,476 C source codes and 46,276 C++ source codes, including 4422 C/C++ header files, for a total of 104,174 files.

### 3.1. BERT-Based Line-Level Weakness Analysis Model

#### 3.1.1. Preparing Training Data

**Data cleanup, normalization, and feature extraction:** Unnecessary comments should be removed within the source code that do not affect code execution or cause weaknesses in the system. In addition, information related to weaknesses should be normalized because the function name contains the sentences “\_bad()” and “\_good()”, which affect the training of the model. In addition, empty lines or areas that were not helpful for training for weakness analysis were deleted to minimize the noise of the model. Through this work, the accuracy of the model can be improved by tokenizing only the necessary information and training.

**Tokenization:** To use a pretrained model, the input data must be transformed into an appropriate format so that each sentence can be sent to a pretrained model to obtain its embedding. We used *neulab/codebert-cpp* [39] pretrained language model to generate a vector



representation of the source code. Here, tokenization is used to make it easier to assign meaning to paragraphs and sentences by dividing them into smaller units, such as in NLP. Typical sub-tokenizer methods include Byte Pair Encoding (BPE) [40], SentencePiece [41], and Word-Piece [42]. Because C/C++ programming languages have a higher complexity of syntax or semantics than other languages, Out-Of-Vocabulary (OOV) situations occur during machine learning. Therefore, in this study, the BPE algorithm in which words are cut into meaningful patterns and tokenized was selected.

**Token and position embedding:** Each token in the source code consists of several tokens that depend heavily on the context (peripheral token) and location between the tokens in the function. Therefore, it is important to capture the code context and its location within the function when predicting the weakness at the function level. The purpose of this step is to perform encoding and embedding that capture the meaning of the code token and its corresponding position in the input sequence (code or line). BERT has three embedding layers: token, sentence, and position embedding [34].

### 3.1.2. Model Training

In this step, the encoding vector input passes through the 12 encoder blocks. Each encoder block consists of two components, a bidirectional multi-head self-attention layer and a fully connected feed-forward neural network. The multi-head self-attention layer is used to calculate the attention weight of a code token that generates a vector of attention, and the token is input in both directions. The final output of the encoder using the neural network preserved the input size of the encoder.

### 3.1.3. Feature Inference

The position information of each token and its relationship to the position were obtained from the output of the BERT model. Subsequently, tokens of each line were collected based on the input source code. We classified the tokens of each reconstructed line into 119 classes (118 CWE has defined in Juliet test suite C/C++ 1.3 version and a class that is not weak) trained through a single linear layer. The attention score may be output to each line of the entire source code through the attention mechanism of BERT. This not only solves the token input limitation problem of BERT but also allows for comparison and evaluation of the attention and static analyzer results output on each line. Table 1 shows an example file tested and printed on the BERT model.

**Table 1.** Example analysis result of BERT-based line-level weakness analysis model.

Line	Source Code	Score
1	void CWE476_NULL_Pointer_Dereference__int_01_bad()	0.1
2	{	0
3	int *data;	0.5
4	/* Set data to NULL */	0
5	data = NULL;	0.63
6	/* POTENTIAL FLAW: Attempt to use data, which may be NULL */	0
7	printf("%d\n", *data);	0.85
8	}	0
9	...	

The score indicates the relationship with other lines, based on lines that are likely to cause weaknesses. In other words, the higher the score, the higher the probability of weakness in the line. In the example shown in Table 1, the line where weakness occurred was analyzed at line 7. Lines 3 and 5 had a command to declare the variable “data”; thus, the score was higher. For annotations or braces, the score was zero and weaknesses could not occur.

### 3.2. Select Multiple Static Analysis Tools and Experiment

In this study, we used some of the following criteria to select multiple static analysis tools: whether open source or license is supported for research, can output or support definition of CWE information; whether data sets such as Juliet test suite C/C++ 1v3 can be tested; whether results and evaluations are good in previous research experiments; and whether document for testing tools are specifically provided. Based on the criteria we selected six tools, namely Cppcheck (CPP) [24], Clang Static Analyzer (CLG) [25], Flawfinder (FLF) [26], Infer (IFR) [28], Frama-C (FRC) [31], and PVS-Studio (PVS) [30]. Table 2 shows the abbreviations, versions, weakness analysis techniques, CWEs-related output information, and the number of CWEs that can be analyzed.

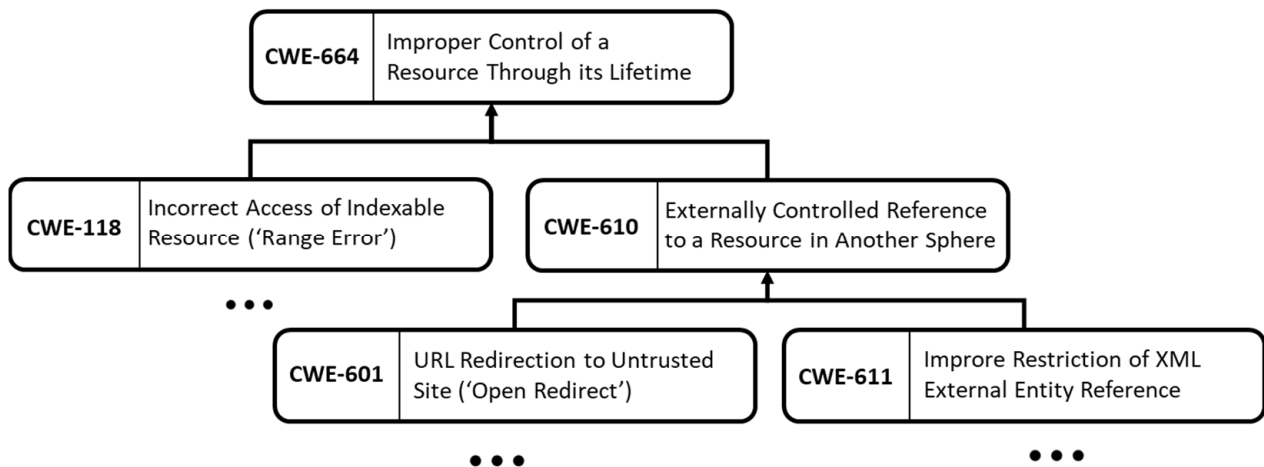
**Table 2.** Overview of selected multiple static analysis tools.

Tool Name & Version	Static Analysis Method	Output CWE	No. CWE
Cppcheck (CPP) 2.8	Provides unique code analysis to detect bugs and focuses on detecting undefined behavior and dangers.	Output	55
Clang Static Analyzer (CLG) 9.0	Path-sensitive, inter-procedural analysis based on symbolic execution.	Matching	57
Flawfinder (FLF) 2.0.19	Using a built-in database of C/C++ functions with well-known problems	Output	17
Infer (IFR) 1.3	Using separation logic, and bi-abduction	Matching	10
Frama-C (FRC) 25.0	Runtime-error detection, value analysis, dependency analysis, and slicing	Matching	35
PVS-Studio (PVS) 7.16	Abstract syntax tree, pattern-based analysis, data-flow analysis, symbolic execution, and taint analysis	Output	114

#### 3.2.1. CWE Mapping and Group

For several weakness static analysis tools, the CWE report describes the relationship between unique identifiers and other types of weakness. However, different static analyses use different identifiers for the types of weaknesses that they support. Due to these different identifiers, it is difficult to automatically evaluate whether the static analysis tool refers to the correct type of weakness, which requires a more rigorous evaluation. As a result, we developed a mapping module to match each analyzer's weakness identifier to the corresponding CWE-ID; the output message for the weakness analyzed for each tool was defined based on CWE-ID. The weakness static analysis tools selected in this study can analyze 157 CWEs with differences in each tool.

In the CWE system, the relationship of each CWE was designated as a child–parent hierarchy. This implies that a child CWE indicates a more specific instance of software weakness in the parent CWE. For instance, CWE-664 (Improper Control of a Resource Through its Lifetime) can be considered as a common denominator for all child CWEs (CWE-118, CWE-610, etc.). Group study of CWEs using child–parent relationships is necessary since only particular CWEs can be evaluated or output results may differ. Even if the exact CWE is not analyzed, the analysis is considered successful if the related CWE is analyzed. In this study, we created a CWE group referring to the CWE VIEW Research concept [43]. Figure 2 shows an example of the structure of the CWE group.



**Figure 2.** Example of CWE group with CWE-664.

### 3.2.2. Tools Experiment

Tools configuration in Figure 1 shows the structure of configuration for integrating analysis results of multiple static analysis tools. In this study, we experimented on Ubuntu, the Juliet test suite C/C++ 1.3 version [38]; the dataset expected in Section 3 was used, and all tools were tested using the command line. There are differences in the required configuration files or options for each tool. The Infer tool was used to compile the entire test case through the prepared “Makefile” and then analyze weaknesses. The Clang Static Analyzer was executed by adding the option “-analyzer-checker = alpha,core”. The Cppcheck and Frama-C tools only passed the path of the test case and the resulting output file, whereas the PVS-Studio tool generated the object file for each test case through “Makefile” compilation and passed the config file “PVS-Studio.cfg”. Because the output results of each tool are different, CWE mapping and groups were conducted for each tool’s analyzed results, and all results were written in one file based on CWE-ID. Table 3 shows the results analyzed for each test case file in one CWE class. In one test case file, multiple tools may analyze the same results but may have different analysis results. One tool can be analyzed as a weakness with multiple lines in one test case. Perhaps the results of the analysis contain a number of several false alarms.

**Table 3.** Example analysis result of weakness multiple static analysis tools with CWE-476.

File Name	Tool Name	Line	CWE-ID	Alarm Message
/NULL_Pointer ... _01.c	CPC	7	476	nullPointer ...
/NULL_Pointer ... _01.c	CLG	7	476	[core.NullDereference]
/NULL_Pointer ... _01.c	IFR	7	476	pointer
/NULL_Pointer ... _01.c	FRC	7	476	warning
/NULL_Pointer ... _01.c	PVS	7	476	the num pointer
/NULL_Pointer ... _01.c	FLF	18	476	security-related
/NULL_Pointer ... _01.c	PVS	18	476	the num pointer
/NULL_Pointer ... _02.c	IFR	10	476	pointer
/NULL_Pointer ... _02.c	CLG	10	476	[core.NullDereference]
/NULL_Pointer ... _03.c	CPC	15	476	nullPointer ...



### 3.3. Weakness Static Analysis Alarm Classification

An experimental environment for multiple static analyses was established, and when the training of the BERT-based line-level weakness analysis model was completed, a dataset was created. As shown in Table 4, the dataset has output analyzed by multiple static analysis tools (1: detected, 0: undetected), BWA (score) model on all test case files in each CWE and metadata provided by Juliet test suite C/C++ 1.3 version (present as a target through confirming the line of weakness which is defined in each test case file).

**Table 4.** Example data of integrating analysis results from each tool for Alarm Classification model.

File Name	Line	CPC	CLG	FLF	IFR	FRC	PVS	BWA	Target
/NULL_Pointer ... _01.c	7	1	1	0	1	1	1	0.85	1
/NULL_Pointer ... _01.c	18	0	0	1	0	0	1	0.25	0
/NULL_Pointer ... _02.c	10	0	1	0	1	0	0	0.35	0
/NULL_Pointer ... _03.c	15	1	0	0	0	0	0	0.1	0

In Table 4, at line 7, 5 out of 6 tools were analyzed through the multiple static analysis tools. The score obtained through the BWA model was 0.85 scores, and the line is likely to have a normal weakness (true alarm or true positive). Thus, it is possible to determine whether it is a true or a false alarm based on the results of multiple static analysis tools on a specific line and the scores obtained through the BWA model.

In this study, this dataset was trained using the decision tree model [44]. Thus, the reliability of the weakness analysis results generated by several static analysis tools can be determined, and false alarms can be reduced by reclassifying the alarm of the weakness analyzer into a decision tree model based on the derived reliability.

## 4. Experimental Setup and Evaluating the Proposed Model

In this study, two deep learning models are used; therefore, it is necessary to separate the Juliet test suite C/C++ 1.3 version dataset used for the first time. A total of 60% and 40% of datasets were split for the BWA model and ACM. Then, for each model, the resulting data were divided into training, validation, and testing data. To train the model, we used PyTorch 1.12.1 with CUDA 11.4 on top of Python 3.10, a machine with 1 Terabyte RAM and 8 NVIDIA Tesla V100 PCIe 32 GB.

To evaluate the performance of the proposed model, scores were calculated for Precision, Accuracy, F1-Score, and Receiver Operating Characteristic area under the curve (AUC). Table 5 shows the evaluation of the BERT-based Line-level weakness analysis and analysis alarm classification model. For the final model, we report several evaluation metrics, including those used in the initial work for each dataset. The metrics are true alarm (true positive), false alarm (false positive), and false alarm rate (false positive rate). This way, we can perform a comparison; the comparison results are shown in Table 6.

**Table 5.** Evaluation of BERT-based Line-level Weakness Analysis and Alarm Classification model.

	Precision	Accuracy	F1-Score
BERT-based Line-level Weakness Analysis model	0.96	0.92	0.94
	F1-Score	Accuracy	AUC
Alarm Classification model	0.89	0.96	0.89

**Table 6.** Experimental results of proposed method for each CWE.

CWE ID	CWE Name	Total Weakness	Analysis Result with Static Analysis Tool					Selected Alarm					Rejected Alarm		Reduce False Alarm Rate **
			Total Alarm	True Alarm	False Alarm	False Alarm Rate *	Total Weakness	Total Alarm	True Alarm	False Alarm	False Alarm Rate	Total Weakness	Rejected Alarm	True Alarm	
126	Buffer Over-read	195	1224	263	961	79%	144	300	235	65	22%	135	924	28	71%
134	Uncontrolled Format String	93	962	135	827	86%	88	211	117	94	45%	79	751	18	74%
195	Signed to Unsigned Conversion Error	207	1984	155	1829	92%	129	215	135	80	37%	119	1769	20	87%
369	Divide by Zero	108	1285	68	1217	95%	36	115	47	68	59%	31	1170	21	88%
401	Memory Leak	275	1839	200	1639	89%	150	237	165	72	30%	141	1602	35	83%
415	Double Free	161	872	97	775	89%	87	210	94	116	55%	81	662	3	75%
416	Use After Free	63	247	82	165	67%	53	72	69	3	4%	49	175	13	60%
457	Use of Uninitialized Variable	220	457	34	423	93%	29	65	33	32	49%	23	392	1	85%
467	Use of sizeof on Pointer Type	9	30	18	12	40%	9	18	16	2	11%	8	12	2	27%
476	Null Pointer Dereference	42	324	62	262	81%	24	61	56	5	8%	24	263	6	77%
563	Unused Variable	69	321	10	311	97%	10	45	9	36	80%	9	276	1	85%
590	Free Memory Not on Heap	189	1239	197	1042	84%	146	275	187	88	32%	137	964	10	76%
676	Use of Potentially Dangerous Function	4	17	4	13	76%	4	4	4	0	0%	4	13	0	76%
688	Function Call with Incorrect Variable	5	31	13	18	58%	5	12	12	0	0%	4	19	1	55%
690	Null Dereference from Return	219	703	131	572	81%	118	136	120	16	12%	111	567	11	78%
758	Undefined Behavior	102	297	36	261	88%	36	65	33	32	49%	28	232	3	76%
762	Mismatched Memory Management Routines	256	1388	357	1031	74%	230	353	321	32	9%	221	1035	36	69%

Table 6 shows the experimental results for each CWE using the proposed method. The contents of Table 6 are CWE-ID, CWE name, total weakness of each CWE, the analysis result with static analysis tools, selected alarm (number of alarms that ACM classified true alarms), rejected alarm (number of alarms that ACM classified false alarms), and reduce false alarm rate (rate of false positive has reduced). As shown in Table 6, the false positive rate (\*) is initially high based on tool analysis results, but the false positive rate (\*\*) is reduced significantly by detecting and excluding false alarms through the proposed Alarm Classification model. For instance, based on all the test case files of Buffer Over-read (CWE-125) weakness, there are 195 weaknesses. Through analysis by static analysis tools, a total of 1224 alarms were generated but only 263 of these were true alarms and the remaining 961 were false alarms. The calculated false alarm rate was 79%, and a total of 144 weaknesses were analyzed. The false alarm rate can be considered high. The proposed model was implemented to classify 1224 alarms, of which 300 were true alarms and 924 were false alarms. However, among the classification results, there are several results that are incorrectly classified (it is a true alarm but classified false alarm: 28, and the other way: 65), and the false alarm rate of the alarm classification model can be calculated as 7%. The false alarm rate of 79% is significantly reduced by 71% when we look at the final results. The overall false detection rate has significantly dropped, despite there being differences in the false alarm rate, which is reduced in accordance with the learning outcomes of each CWE.

## 5. Conclusions and Further Research

In the era of the fourth Industrial Revolution, the importance of software has recently emerged in many industrial fields. As a result, software reliability and safety have become more important because of the resource protection problem of software, and software development security has become an important factor in software development.

Static analysis is a code analysis technique that has been developed to analyze weaknesses in source code due to the limits of manually analyzing weaknesses in software development. However, static analysis alone might not be able to detect weaknesses, and misuse may occur if it is approached conservatively. Especially, since each static analysis tool has a unique set of diagnostic weaknesses and analysis capabilities, when developing software, several static analysis tools are frequently used, and as the volume of the source code increases it leads to more false positives.

In this study, we present a technique that can effectively analyze weaknesses and reduce false detections using the advantages of deep learning while maintaining the advantages of static analysis tools. The BERT model is used in this model to analyze syntactic and semantic weaknesses in the challenging C/C++ source code. The evaluation results of the model showed a precision of 96%, an accuracy of 92%, and an F1-Score of 94%. In addition, the possible score of weaknesses for each line can be calculated using the self-attention mechanism of the proposed model. This model determines the reliability of the weakness analysis results generated by several static analysis tools and reclassifies the derived results into a decision tree model. The trained decision tree model receives scores obtained for each line and several static analysis results for the C/C++ source code through the BERT model to determine whether each alarm is true or false. When evaluating this model, an accuracy of 89%, an F1-Score of 96%, and an AUC of 0.89 were derived.

As a result, many false alarms are identified and minimized using the proposed methods, while some false alarms are identified as true alarms from the results of the weakness static analysis. Examining for typos can save developers time and money that might otherwise be wasted. Additionally, by eliminating security weaknesses during the software development process, it can significantly improve the security of the software project. In the future, we intend to improve the false alarm reduction model by adding features to the alarm classification model.

**Author Contributions:** Conceptualization, D.H.N., A.S. and Y.S.; software, D.H.N. and N.P.N.; validation, A.S. and Y.S.; formal analysis, D.H.N. and Y.S.; investigation, D.H.N. and N.P.N.; resources,

D.H.N.; data curation, D.H.N. and N.P.N.; writing—original draft preparation, D.H.N. and A.S.; writing—review and editing, N.P.N. and Y.S.; visualization, D.H.N. and A.S.; supervision, A.S. and Y.S.; project administration, Y.S. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Not applicable.

**Acknowledgments:** This work was supported by the Korea Institute of Energy Technology Evaluation and Planning (KETEP) and the Ministry of Trade, Industry & Energy (MOTIE) of the Republic of Korea (No. 20224000000020). The research was supported by the National Research Foundation of Korea (NRF) grant by the Korean government (MSIT) (No. 2018R1A5A7023490).

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

## References

1. CS Hub Mid-Year Market Report 2022 | Cyber Security Hub. Available online: <https://www.cshub.com/executive-decisions/reports/cs-hub-mid-year-market-report-2022> (accessed on 7 November 2022).
2. Jeong, J.; Son, Y.; Lee, Y. A Study on the Secure Coding Rules for Developing Secure Smart Contract on Ethereum Environments. *Int. J. Adv. Sci. Technol.* **2019**, *133*, 47–58. [CrossRef]
3. Kim, S.; Yunsik Son, Y.L. A Study on the Security Weakness Analysis of Chaincode on Hyperledger Fawbric and Ethereum Blockchain Framework. *J. Green Eng.* **2020**, *10*, 6349–6367.
4. Jeong, J.; Joo, J.W.J.; Lee, Y.; Son, Y. Secure Cloud Storage Service Using Bloom Filters for the Internet of Things. *IEEE Access* **2019**, *7*, 60897–60907. [CrossRef]
5. Jang-Jaccard, J.; Nepal, S. A Survey of Emerging Threats in Cybersecurity. *J. Comput. Syst. Sci.* **2014**, *80*, 973–993. [CrossRef]
6. Zaharia, S.; Rebedea, T.; Trausan-Matu, S. Machine Learning-Based Security Pattern Recognition Techniques for Code Developers. *Appl. Sci.* **2022**, *12*, 12463. [CrossRef]
7. 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*, 1692. [CrossRef]
8. Li, Z.; Zou, D.; Tang, J.; Zhang, Z.; Sun, M.; Jin, H. A Comparative Study of Deep Learning-Based Vulnerability Detection System. *IEEE Access* **2019**, *7*, 103184–103197. [CrossRef]
9. Ziems, N.; Wu, S. Security Vulnerability Detection Using Deep Learning Natural Language Processing. In Proceedings of the IEEE INFOCOM 2021—IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), Vancouver, BC, Canada, 10–13 May 2021. [CrossRef]
10. Li, Z.; Zou, D.; Xu, S.; Jin, H.; Zhu, Y.; Chen, Z. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* **2018**, *19*, 2244–2258. [CrossRef]
11. Li, Z.; Zou, D.; Xu, S.; Ou, X.; Jin, H.; Wang, S.; Deng, Z.; Zhong, Y. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. *arXiv* **2018**, arXiv:1801.01681.
12. Zhou, Y.; Liu, S.; Siow, J.; Du, X.; Liu, Y. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. *Adv. Neural Inf. Process. Syst.* **2019**, *32*, 10197–10207.
13. Most Secure Programming Languages—Mend. Available online: <https://www.mend.io/most-secure-programming-languages/> (accessed on 16 October 2022).
14. Which Industries Use C++? | Snyk. Available online: <https://snyk.io/learn/who-uses-cpp/> (accessed on 7 November 2022).
15. Git. Available online: <https://git-scm.com/> (accessed on 7 November 2022).
16. OpenSSH. Available online: <https://www.openssh.com/> (accessed on 7 November 2022).
17. Download PuTTY—A Free SSH and Telnet Client for Windows. Available online: <https://www.putty.org/> (accessed on 7 November 2022).
18. FileZilla—The Free FTP Solution. Available online: <https://filezilla-project.org/> (accessed on 7 November 2022).
19. CWE—Common Weakness Enumeration. Available online: <https://cwe.mitre.org/> (accessed on 9 December 2022).
20. CAPEC—Common Attack Pattern Enumeration and Classification (CAPECTM). Available online: <https://capec.mitre.org/> (accessed on 9 December 2022).
21. CVE—CVE. Available online: <https://cve.mitre.org/> (accessed on 9 December 2022).
22. CWE—2022 CWE Top 25 Most Dangerous Software Weaknesses. Available online: [https://cwe.mitre.org/top25/archive/2022/2022\\_cwe\\_top25.html](https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html) (accessed on 9 December 2022).
23. OWASP Top Ten | OWASP Foundation. Available online: <https://owasp.org/www-project-top-ten/> (accessed on 9 December 2022).

24. Cppcheck—A Tool for Static C/C++ Code Analysis. Available online: <https://cppcheck.sourceforge.io/> (accessed on 7 November 2022).
25. Clang Static Analyzer. Available online: <https://clang-analyzer.lvm.org/> (accessed on 7 November 2022).
26. Flawfinder Home Page. Available online: <https://dwheeler.com/flawfinder/> (accessed on 7 November 2022).
27. CodeQL. Available online: <https://codeql.github.com/> (accessed on 9 December 2022).
28. Infer Static Analyzer | Infer | Infer. Available online: <https://fbinfer.com/> (accessed on 7 November 2022).
29. Code Quality and Code Security | SonarQube. Available online: <https://www.sonarqube.org/> (accessed on 8 December 2022).
30. PVS-Studio Is a Solution to Enhance Code Quality, Security (SAST), and Safety. Available online: <https://pvs-studio.com/en/> (accessed on 7 November 2022).
31. Frama-C—Framework for Modular Analysis of C Programs. Available online: <https://frama-c.com/> (accessed on 7 November 2022).
32. Jeong, J.; Lim, J.Y.; Son, Y. A Data Type Inference Method Based on Long Short-Term Memory by Improved Feature for Weakness Analysis in Binary Code. *Future Gener. Comput. Syst.* **2019**, *100*, 1044–1052. [CrossRef]
33. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention Is All You Need. *Adv. Neural Inf. Process. Syst.* **2017**, *2017*, 5999–6009. [CrossRef]
34. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Minneapolis, MN, USA, 2–7 June 2018; Volume 1, pp. 4171–4186. [CrossRef]
35. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *arXiv* **2020**, arXiv:2002.08155.
36. Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. GraphCodeBERT: Pre-Training Code Representations with Data Flow. *arXiv* **2020**, arXiv:2009.08366.
37. Wang, Y.; Wang, W.; Joty, S.; Hoi, S.C.H. CodeT5: Identifier-Aware Unified Pre-Trained Encoder-Decoder Models for Code Understanding and Generation. *arXiv* **2021**, arXiv:2109.00859.
38. Black, P.E. *Juliet 1.3 Test Suite: Changes From 1.2*; US Department of Commerce, National Institute of Standards and Technology: Gaithersburg, MD, USA, 2010. [CrossRef]
39. Neulab/Code-Bert-Score: CodeBERTScore: An Automatic Metric for Code Generation, Based on BERTScore. Available online: <https://github.com/neulab/code-bert-score> (accessed on 28 December 2022).
40. Bostrom, K.; Durrett, G. Byte Pair Encoding Is Suboptimal for Language Model Pretraining. *arXiv* **2004**, arXiv:2004.03720.
41. Kudo, T.; Richardson, J. SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing. *arXiv* **2018**, arXiv:1808.06226.
42. Wu, Y.; Schuster, M.; Chen, Z.; Le, Q.V.; Norouzi, M.; Macherey, W.; Krikun, M.; Cao, Y.; Gao, Q.; Macherey, K.; et al. Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv* **2016**, arXiv:1609.08144.
43. CWE—CWE-1003: Weaknesses for Simplified Mapping of Published Vulnerabilities (4.9). Available online: <https://cwe.mitre.org/data/definitions/1003.html> (accessed on 7 November 2022).
44. 1.10. Decision Trees—Scikit-Learn 1.2.0 Documentation. Available online: <https://scikit-learn.org/stable/modules/tree.html> (accessed on 27 December 2022).

**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.