electronics

MDPI

Article

AdVulCode: Generating Adversarial Vulnerable Code against Deep Learning-Based Vulnerability Detectors

Xueqi Yu, Zhen Li *, Xiang Huang and Shasha Zhao

School of Cyber Security and Computer, Hebei University, Baoding 071002, China
* Correspondence: zh_li@hust.edu.cn

Abstract: Deep learning-based vulnerability detection models have received widespread attention; however, these models are susceptible to adversarial attack, and adversarial examples are a primary research direction to improve the robustness of the models. There are three main categories of adversarial example generation methods for source code tasks: changing identifier names, adding dead code, and changing code structure. However, these methods cannot be directly applied to vulnerability detection. Therefore, we propose the first study of adversarial attack on vulnerability detection models. Specifically, we utilize equivalent transformations to generate candidate statements and introduce an improved Monte Carlo tree search algorithm to guide the selection of candidate statements to generate adversarial examples. In addition, we devise a black-box approach that can be applied to widespread vulnerability detection models. The experimental results show that our approach achieves attack success rates of 16.48%, 27.92%, and 65.20%, respectively, in three vulnerability detection models with different levels of granularity. Compared with the state-of-the-art source code attack method ALERT, our method can handle models with identifier name mapping, and our attack success rate is 27.59% higher on average than ALERT.

Keywords: vulnerability detection; adversarial examples; code transformation; deep learning

check for updates

Citation: Yu, X.; Li, Z.; Huang, X.; Zhao, S. AdVulCode: Generating Adversarial Vulnerable Code against Deep Learning-Based Vulnerability Detectors. *Electronics* **2023**, *12*, 936. https://doi.org/10.3390/electronics12040936

Academic Editor: Aryya Gangopadhyay

Received: 13 January 2023
Revised: 1 February 2023
Accepted: 10 February 2023
Published: 13 February 2023

1. Introduction

Software vulnerabilities are prevalent, as evidenced by the steady increase of vulnerabilities reported by the Common Vulnerabilities and Exposures (CVE) [1]. The ideal solution is to detect and patch security problems before the software is released. Static analysis [2–6], which detects vulnerabilities by analyzing code without software execution, is widely used because of its high coverage of code paths and high efficiency. In parallel, dynamic analysis [7,8], which executes the software while monitoring its behavior, is also an effective way because of its high accuracy in discovering vulnerabilities. In this paper, we center on static analysis-based vulnerability detectors. A recent development in static analysis-based vulnerability detection is the integration of Deep Learning (DL) techniques. DL-based vulnerability detectors [9–13] have attracted much attention because they do not require human experts to define vulnerability features and can achieve low false positive rates and low false negative rates.

Adversarial examples against DL models have been studied in many domains, such as image processing [14–17], automatic speech recognition [18–21], natural language processing [22–24], and source code processing [25–28]. Intuitively, there is also a significant hazard lying under the DL models in vulnerability detection, i.e., the lack of adversarial robustness. From the perspective of human beings, the vulnerable code often has small differences from the non-vulnerable examples but is incorrectly classified as non-vulnerable by DL-based vulnerability detectors. Such vulnerable code evades vulnerability detectors and can be used to launch damaging attacks, for example, using the hidden vulnerability as a backdoor to conduct malicious behaviors [29]. However, the adversarial examples against DL-based vulnerability detectors have not been investigated. Existing approaches to generating adversarial examples for other domains are incompetent for vulnerability detection owing to the following three reasons.

First, unlike the continuous space of images and speeches, vulnerability detection has a discrete space of source code, which is difficult to optimize. Moreover, small perturbations of source code are often clearly perceptible; the replacement of even a character in a statement may lead to incorrect syntax and failure to compile. Second, unlike the loose constraints of natural languages, the source code for vulnerability detection is strictly bounded by rigid lexical and syntactical constraints. Therefore, the adversarial perturbations of source code must satisfy all these constraints; otherwise, the generated adversarial examples would encounter compilation errors and would be directly rejected for further analysis. Third, the approaches to generating adversarial examples in source code processing [25–28], including identifier renaming [27,28,30], dead-code insertion [28], and code structure changes [25,26], are incompetent for vulnerability detection because (i) identifier renaming is ineffective for many vulnerability detectors, which usually maps the identifier to symbolic names; (ii) dead-code insertion (for example, inserting a new unused variable declaration) is ineffective for program slice-based vulnerability detectors because the inserted dead code is independent of vulnerable statements and would not be included in the program slices for vulnerability detection; and (iii) code structure changes usually involve large changes, not small perturbations.

Our aims. DL-based vulnerability detection models still have the problem of insufficient robustness, and these models are susceptible to adversarial attacks. However, the main research to improve the robustness of models is the generation of adversarial examples. This paper aims to generate adversarial samples for DL-based vulnerability detection models that differ in operator granularity, vector representation, and neural network structure.

Our contributions. In this paper, we present ADVULCODE, the first framework for the efficient generation of adversarial example code for DL-based vulnerability detection models. The contributions of this paper include:

- We present the first automated attack framework for vulnerability detection models, which is designed as a black-box that can work for a wide range of vulnerability detection models;
- We devise a transformation strategy to generate candidate lists using a combination of applying different transformation rules and repeating a single rule. This transformation strategy is effective in ensuring the quality and quantity of candidate statements;
- We improve the Monte Carlo tree search (MCTS) algorithm used in guiding the generation of adversarial examples to enhance the efficiency of adversarial example generation;
- We select three vulnerability detection models with different deep neural networks and different levels of granularity, i.e., slice + token + BGRU, function + token + BLSTM, and function + graph + GGNN, and our attack success rate reaches 16.48%, 27.92%, and 65.20%, respectively. Moreover, we further investigate the impact of transformation rules and vulnerability lines on the generation of adversarial examples.

The remainder of this paper is organized as follows: Section 2 discusses the basic framework of the DL-based vulnerability detectors. Section 3 discusses our selected code transformation rules. In Section 4, we present the design of our attack; Section 5 presents our experiments and results; Section 6 discusses limitations of the present study; Section 7 describes our future work; Section 8 reviews the related prior work; and Section 9 concludes the present paper.

2. DL-Based Vulnerability Detection

Figure 1 illustrates the process of DL-based vulnerability detection. There are two phases: the training phase and the detection phase. In the training phase, the input is the source code of training programs for training a DL model, and the output is the trained DL model. In the detection phase, the input is the source code of target programs for vulnerability detection, and the output is the classification results of code examples in target programs, i.e., the class and the probability that the DL model predicts the code example as vulnerable. The training phase involves Steps 1, 2, and 3, and the detection phase involves Steps 1, 2, and 4.

- Step 1: Generating code examples. We decompose training programs and target programs into code examples. Each code example can be a function [11,13,31] or a program slice [9,10,32], where a program slice is a number of statements that are semantically related to each other in terms of data dependency and control dependency. The corresponding DL-based vulnerability detectors are respectively called function-level and slice-level vulnerability detectors;
- Step 2: Transforming code examples into vectors. In order to input the code examples into the DL model, each code example from training and target programs needs to be represented as a sequence of tokens [9,10,32] or Abstract Syntax Trees (ASTs) [11,33,34], or graphs [13], and then encoded into vectors. For training programs, the vector corresponding to each code example is labeled as "1" (i.e., vulnerable) or "0" (i.e., non-vulnerable) according to whether it is vulnerable or not;
- Step 3: Training a DL model. We leverage the vectors corresponding to code fragments and their labels from training programs to learn a deep neural network, e.g., Bidirectional Gated Recurrent Unit (BGRU) [10], Bidirectional Long Short-Term Memory (BLSTM) [9,33], and Gated Graph Neural Networks (GGNN) [13];
- Step 4: Applying the trained deep learning model to detect vulnerabilities. We apply the trained DL model in the detection phase to detect vulnerabilities in target programs, and output the classification results of code examples in target programs.

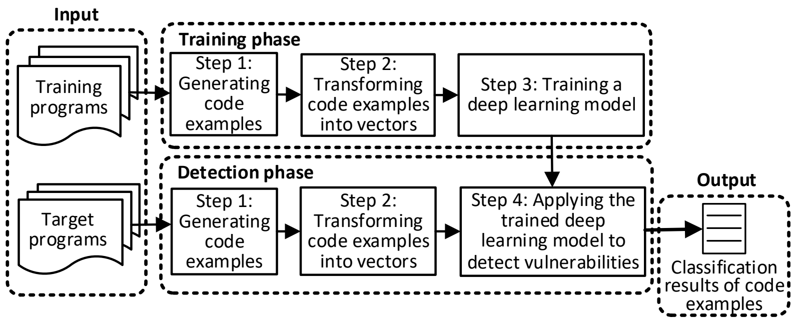

Input
Training programs
Target programs
Training phase
Step 1: Generating code examples
Step 2: Transforming code examples into vectors
Step 3: Training a deep learning model
Detection phase
Step 1: Generating code examples
Step 2: Transforming code examples into vectors
Step 4: Applying the trained deep learning model to detect vulnerabilities
Output
Classification results of code examples

Figure 1. Process of DL-based vulnerability detection.

3. Code Transformation Rules

In this section, we introduce three types of code-equivalent transformations, containing 13 transformation rules. Our transformations focus on syntactic and variable elements. The purpose of the code transformations is to add perturbations to the code statements to generate a list of candidate adversarial statements. These candidate adversarial statements in different code statements can be combined to generate the final adversarial examples.

In the domains of natural language processing and images, the feature space is continuous, while for code, the feature space is discrete. Therefore, there are different requirements in the source code domain. Specifically, we consider the following three factors when choosing code transformations. The first factor is preserving semantics. Semantics-preserving transformations ensure that the functionality of the code is unchanged before and after the transformation. We consider code transformations to be meaningful only if the functionality of the code is unchanged. The second factor is syntactic correctness. The perturbed code should be syntactically correct. Otherwise, the program will encounter compilation errors that are easily detectable. The third factor is preserving vulnerability. Unlike other source code processing tasks, the preservation of vulnerabilities in vulnerability detection is necessary. Vulnerable programs should still preserve vulnerabilities after code perturbation. Otherwise, the code example that makes the vulnerability detection model fail is not an adversarial example because the true vulnerability label has changed.

In short, we select code transformations based on these three factors. We have classified these 13 different transformations into three types, which are briefly introduced below.

3.1. Operator Transformations

As shown in Table 1, operator transformations change operators in code, such as relational and subscript operators, into forms that are semantically identical. The replaceable token (Rule 1) is only available in C++ and contains a total of 11 pairs of replaceable tokens. The relational operator transformation (Rule 2) changes the expression of relations, but not the logical result; therefore, this rule allows more flexibility in changing specific tokens. We also add a perturbation strength parameter to allow multiple negation transformations for relational operator transformations.

Table 1. Operator transformation rules

ID	Rule Type	Rule Description
1	Replaceable token	Replace with replaceable token identifier or operator only C++, e.g., && $\rightarrow$ *and*.
2	Relational operator transformation	Change relational operators, swapping variables as needed, e.g., $a > b \rightarrow b < a$ *or* $!(a <= b)$.
3	Array usage transformation	Transforming the way to reference array elements, e.g., $a[1] = 1 \rightarrow *(a+1) = 1$.
4	Equivalent calculation	Change the arithmetic operator, e.g., $a + b \rightarrow a - (-b)$.
5	Increment and decrement operator transformation	Transforming increment and decrement operator in logically independent statements, e.g., $index = index + 1, ++index$ *or* $index += 1$.
6	Pointer-reference transformation	Transforming the reference way of data, e.g., $a.b \rightarrow \&a->b$.

3.2. Data Transformations

Data transformations mainly change the data type or data representation. As shown in Table 2, the static array definition (Rule 7) can be replaced by the dynamic `malloc` method. Using `typeof` expression, (Rule 8) can replace the type name of a variable "`a`" with "`typeof(b)`", assuming that "`b`" and "`a`" are the same type and that the variable "`b`" appears before "`a`". We can also split the value of an integer literal into two numbers multiplied together or convert it into other numerical bases (Rule 10).

Table 2. Data transformation rules

ID	Rule Type	Rule Description
7	Array definition	Transforming the way an array is defined, e.g., $a = [b] \rightarrow a = malloc(b * size(a))$.
8	Using `typeof` expression	Use `typeof` to dynamically generate data types, e.g., $int\ a = b \rightarrow typeof(b)\ a = b$.
9	Casting transformation	Cast variables into certain data types, e.g., $int\ a = b \rightarrow int\ a = (int)\ b$.
10	Integer transformation	Transforming an integer literal to expression or hex, e.g., $int\ a = 8 \rightarrow int\ a = 4 * 2$.
11	Character representation	Transforming character literals to ASCII code, e.g., $'a' \rightarrow 97$.

3.3. Bogus Code Transformations

Besides the operator transformations and data transformations described above, bogus code transformations within expressions are also considered. As shown in Table 3, unlike the traditional addition of dead code, bogus code works inside expressions and provides a distraction while not affecting the normal function of the code. For example, adding useless calculation expression (Rule 13) introduces some temporary variables that finally evaluate to zero.

Table 3. Bogus code transformation rules

ID	Rule Type	Rule Description
12	Adding useless multiplier	Multiply an expression by a random number and then dividing the result by that number, e.g., $a * d \rightarrow a * d * num/num$; where *num* is a random number.
13	Adding useless calculation expression	Add useless expression to multiplication and subtraction operations, e.g., $a + b \rightarrow a + b + expr$; where *expr* is an expression whose result is 0.

4. Attack Design

4.1. Problem Formulation

Given a DL-based vulnerability detector M and a target program p_i, the output of M is a set of code examples $X = \{x_{i,1}, \ldots, x_{i,n}\}$ generated from p_i with the predicted class $M(x_{i,j})$ for each $x_{i,j}$, where $1 \leq j \leq n$ and $M(x_{i,j}) \in \{0, 1\}$ ("0" means non-vulnerable and "1" means vulnerable). The attacker aims to generate an adversarial program p'_i from p_i, which contains some adversarial code example $x'_{i,j}$ (generated from p'_i) whose ground truth label is "1" (i.e., vulnerable), but whose $M(x'_{i,j}) = 0$ (i.e., non-vulnerable).

Adversarial attacks can be divided into two categories: (i) white-box attack allows full access to the subject model, including outputs, gradients and hyper-parameters; (ii) black-box attack only allows the attackers to have access to model outputs. Because the black-box attack does not require internal knowledge of the DL-based vulnerability detectors, it is applicable to any learning algorithm and suitable for evading a wide range of vulnerability detectors. In this paper, we focus on the black-box setting.

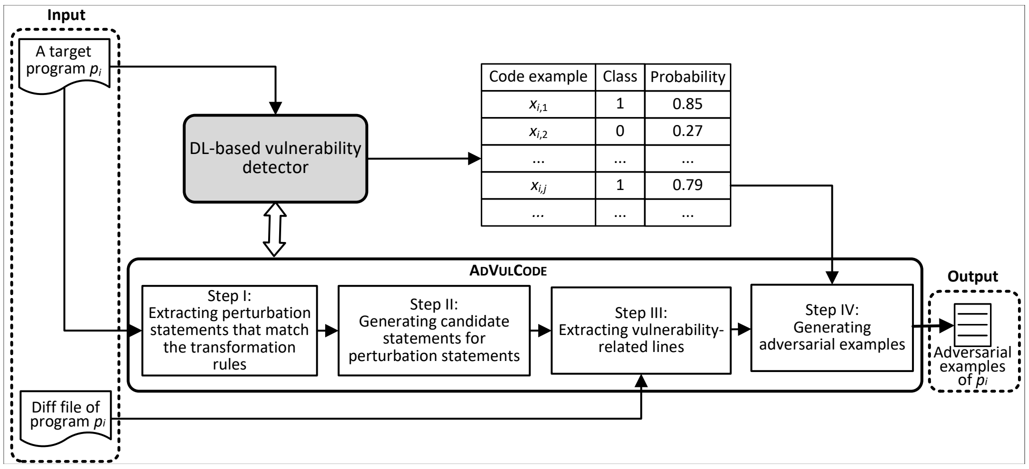
4.2. AdVulCode

Figure 2 illustrates the structure of AdVulCode. The input to AdVulCode is a vulnerable program p_i and a diff file of the vulnerable code. The output is the adversarial examples of program p_i, whose code examples are all incorrectly predicted by the DL-based vulnerability detector M.

AdVulCode utilises code transformations to generate candidate perturbation statements. Then, we filter out statements that are from non-vulnerability-related lines, and finally, we use the MCTS algorithm to guide the adversarial example generation. We will elaborate on the following four steps in subsequent subsections:

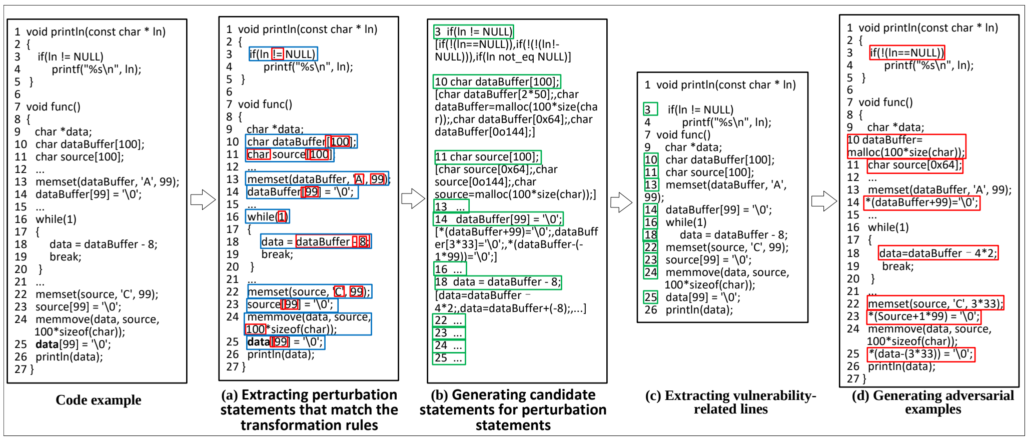
- Step I: Extracting all code statements and locations from the target program p_i that match the conditions of the transformation rules;
- Step II: Generating replaceable candidate statements lists for target code statements;
- Step III: Extracting the lines of code associated with the vulnerable variables in the data flow and control flow. Filter out non-vulnerability-related lines;
- Step IV: Generating adversarial examples using an improved MCTS algorithm.

Figure 3 uses a simple example of vulnerable code to describe the process of the whole attack. Note that the real vulnerable code is far more complex than this.


Input
A target program p_i
DL-based vulnerability detector
ADVULCODE
Step I: Extracting perturbation statements that match the transformation rules
Step II: Generating candidate statements for perturbation statements
Step III: Extracting vulnerability-related lines
Step IV: Generating adversarial examples
Output
Adversarial examples of p_i
Diff file of program p_i

Code example	Class	Probability
$x_{i,1}$	1	0.85
$x_{i,2}$	0	0.27
...	...	...
$x_{i,j}$	1	0.79
...	...	...

Figure 2. General overview of ADVULCODE, where the inputs are source program p_i and diff files, and the output is the adversarial examples of the program p_i.



```
1  void println(const char * ln)
2  {
3     if(ln != NULL)
4         printf("%s\n", ln);
5   }
6
7  void func()
8  {
9    char *data;
10   char dataBuffer[100];
11   char source[100];
12   ...
13   memset(dataBuffer, 'A', 99);
14   dataBuffer[99] = '\0';
15   ...
16   while(1)
17   {
18       data = dataBuffer - 8;
19       break;
20    }
21   ...
22   memset(source, 'C', 99);
23   source[99] = '\0';
24   memmove(data, source,
     100*sizeof(char));
25   data[99] = '\0';
26   println(data);
27 }
```

```
1  void println(const char * ln)
2  {
3     if(ln != NULL)
4         printf("%s\n", ln);
5   }
6
7  void func()
8  {
9    char *data;
10   char dataBuffer[100];
11   char source[100];
12   ...
13   memset(dataBuffer, 'A', 99);
14   dataBuffer[99] = '\0';
15   ...
16   while(1)
17   {
18       data = dataBuffer - 8;
19       break;
20    }
21   ...
22   memset(source, 'C', 99);
23   source[99] = '\0';
24   memmove(data, source,
     100*sizeof(char));
25   data[99] = '\0';
26   println(data);
27 }
```

(a) Extracting perturbation statements that match the transformation rules

```
3  if(ln != NULL)
[if(!(ln==NULL)),if(!(!(ln!-
NULL))),if(ln not_eq NULL)]

10 char dataBuffer[100];
[char dataBuffer[2*50];,char
dataBuffer=malloc(100*size(cha
r));,char dataBuffer[0x64];,char
dataBuffer[0o144];]

11 char source[100];
[char source[0x64];,char
source[0o144];,char
source=malloc(100*size(char));]
13  ...
14  dataBuffer[99] = '\0';
[*(dataBuffer+99)='\0';,dataBuff
er[3*33]='\0';,*(dataBuffer-(-
1*99))='\0';]
16  ...
18  data = dataBuffer - 8;
[data=dataBuffer −
4*2;,data=dataBuffer+(-8);,...]
22  ...
23  ...
24  ...
25  ...
```

(b) Generating candidate statements for perturbation statements

```
1  void println(const char * ln)

3     if(ln != NULL)
4         printf("%s\n", ln);
7  void func()
9    char *data;
10   char dataBuffer[100];
11   char source[100];
13   memset(dataBuffer, 'A',
99);
14   dataBuffer[99] = '\0';
16   while(1)
18       data = dataBuffer - 8;
22   memset(source, 'C', 99);
23   source[99] = '\0';
24   memmove(data, source,
     100*sizeof(char));
25   data[99] = '\0';
26   println(data);
```

```
1  void println(const char * ln)
2  {
3     if(!(ln==NULL))
4         printf("%s\n", ln);
5   }
6
7  void func()
8  {
9    char *data;
10 dataBuffer=
malloc(100*size(char));
11   char source[0x64];
12   ...
13   memset(dataBuffer, 'A', 99);
14  *(dataBuffer+99)='\0';
15   ...
16   while(1)
17   {
18       data=dataBuffer − 4*2;
19       break;
20    }
21   ...
22   memset(source, 'C', 3*33);
23   *(Source+1*99) = '\0';
24   memmove(data, source,
     100*sizeof(char));
25   *(data-(3*33)) = '\0';
26   println(data);
27 }
```

Figure 3. An example illustrating the steps of attack process.

4.2.1. Step I: Extracting Perturbation Statements that Match the Transformation Rules

To extract code statements containing transformation rules, we use the code parsing tool srcML to implement our work. We use srcML for two reasons. Firstly, srcML does not require the code that it parses to be complete (i.e., with all dependencies and headers available), which is friendly to vulnerability datasets because such datasets are usually composed of vulnerability files or vulnerability functions. Secondly, the results of srcML parsing are stored in XML format, which makes it easy to view and change the structure of the code's AST tree and allows for more flexible code transformations.

Algorithm 1 describes the process of extracting statements that match transformation rules. For each program p_i, we use *srcML* to parse it into an AST tree p_{ast} (line 2). We extract the nodes that match the transformation rules by iterating through the AST tree (line 4). For example, to extract statements for which the pointer-reference transformation is applicable, we locate all "`<operator>`" nodes in the AST tree with a value of "`->`" or "`.`", extract the statement nodes, and assign them to the variable `stmts`. In this way, we obtain all statements that match the transformation rules. Finally, we integrate the statement location s_{loc}, the type of the transformation rule t_{type}, and the node location of the transformation rule s_{tar} for these statements (lines 6–11).

Figure 3a illustrates the process of extracting target perturbation statements. The red boxes are points that match the rule features, and the blue boxes are the statements that are the targets of perturbation.

Algorithm 1 Extracting perturbation statements that match the transformation rules

```
Input: $p_i$ (vulnerable program); $T$(the set of code transformation rules); $t_{i\_k}$(rule transformation flags for corresponding rules $t_i$ in rule $T$);
Output: The set $R$ of perturbed code statements and positions in program $p_i$ that match the code transformation rules $T = \{t_1, \dots, t_i\}$.
1: $R \leftarrow \varnothing$; {A set containing the target perturbation code and locations}
2: $p_{ast} \leftarrow srcml(p_i)$; {the AST tree parsed from $p_i$}
3: for each rule $t_i \in T$ do
4:   $stmts \leftarrow$ Get all statements from $p_{ast}$ that match rule $t_i$;
5:   for each statement $s_i \in stmts$ do
6:     if $t_i\_k \in s_i$ then
7:       $t_{type} \leftarrow$ Get the rule type of the $t_i$_k transformation;
8:       $s_{loc} \leftarrow$ Get the location of statement $s_i$;
9:       $s_{tar} \leftarrow$ Get the target node of statement $s_i$;
10:      $R \leftarrow R \cup \{\{s_i + s_{loc} : [t_{type}, s_{tar}]\}\}$;
11:    end if
12:  end for
13: end for
14: return $R$;
```

4.2.2. Step II: Generating Candidate Statements for Perturbation Statements

In this step, after determining the target statements and locations of the perturbation, we need to generate candidate statements for the target statements. We select 13 transformation rules, and for each statement there can be multiple transformation rules applicable. To ensure the quality of the generated adversarial examples, we consider controlling the generation of candidate statements in terms of both breadth and depth. We set two parameters k and m to control the depth and breadth, respectively. Briefly, k means that our transformations will loop k times, and m means that, after we have applied the rule m times, we save the current transformed statement once.

As shown in Algorithm 2, for single-rule transformations, we apply all transformation rules for the target code statement individually (lines 2–9). For multi-rule combination transformations (lines 10–31), we control the generation of perturbation statements by setting the parameters k and m. At each iteration of the transformations, the perturbation target node (line 22) is extracted again based on the transformed statements, and the statements are saved once the rule has been applied m times (line 19), and then the transformation is continued for all the perturbed transformation nodes. The cycle repeats until the whole transformation loop has been iterated k times. The final transformation results are a set of candidate statements (line 26).

Figure 3b illustrates the process of generating candidate statements for the target statements. The green boxes indicate the target statements, followed by a list containing candidate statements.

Algorithm 2 Generating candidate statements for perturbation statements

```
Input: $S$ (statements with transformation type, perturbation position, perturbation node); $k$ (number of iterations of code transformations); $m$ (number of rule tokens for code transformations); $n$ (number of candidate statements)
Output: The set of candidate statements $T$ for the statement set $S = \{S_1, \ldots, S_i\}$
1:  $T \leftarrow \varnothing$;
2:  for each target code statement $s_i \in S$ do
3:    $R_i \leftarrow$ The set of all target transformation rules in $s_i$;
4:    for $r_{i,j} \in R_i$ do
5:      $candidate \leftarrow$ Get the statement after $s_i$ has applied the rule $r_{i,j}$;
6:      $s_{loc} \leftarrow$ Get the location of statement $s_i$;
7:      $T \leftarrow T \cup \{\{s_i + s_{loc} : [candidate]\}\}$;
8:    end for
9:  end for
10: for each target code statement $s_i \in S$ do
11:   while each candidate statement n do
12:     $R_i \leftarrow$ The set of all token transformation rules in $s_i$;
13:     $s_{tmp} \leftarrow s_i$;
14:     $f_m \leftarrow 0$;
15:     while $k > 0$ do
16:       for $r_{i,j} \in R_i$ do
17:         if $m - f_m == 0$ then
18:           $f_m \leftarrow 0$;
19:           $candidate \leftarrow s_{tmp} \cup candidate$ ;
20:         end if
21:         $s_{tmp} \leftarrow$ Get the new $s_{tmp}$ after $s_{tmp}$ has applied the rule $r_{i,j}$;
22:         $R_i \leftarrow$ The set of all token transformation rules in $s_{tmp}$;
23:       end for
24:       $k \leftarrow k - 1$;
25:     end while
26:     $candidate \leftarrow s_{tmp}$ ;
27:     $s_{loc} \leftarrow GetLoc(s_i)$;
28:     $T \leftarrow T \cup \{\{s_i + s_{loc} : [candidate]\}\}$;
29:     $n \leftarrow n - 1$;
30:   end while
31: end for
32: return $T$;
```

4.2.3. Step III: Extracting Vulnerability-Related Lines

The focus of DL-based vulnerability detection models is on vulnerabilities. To locate the lines of code associated with the vulnerability, we choose the same slicing extraction method as in the SySeVR [10] article to extract the lines associated with the vulnerability. In summary, this is achieved by using the modified lines in the diff file. The modified lines in the diff file of the code represent the conditions under which the vulnerability is triggered or the cause of the vulnerability. The modified lines in the diff file are used to locate variables associated with the vulnerability, which we call vulnerability variables. After obtaining the vulnerable lines, we use the Joern code parsing tool to parse the vulnerability code p_i to obtain the data flow and control flow of the vulnerability lines, and to locate the lines of code that have data dependency or control dependency on the vulnerability lines. These lines of code are called vulnerability-related lines.

Figure 3c shows the extracted vulnerability-related lines. Note that the code used here is relatively simple, so the vulnerability-related lines here already include all of the candidate statement lines, but the real situation will be different.

4.2.4. Step IV: Generating Adversarial Examples

In this subsection, we utilize the improved MCTS algorithm to guide the generation of adversarial examples. For each program p_i, we have a set of target statements $S_i \in p_i$, in which there are n target statements $s_{i,j} \in S_i$, and each target statement $s_{i,j}$ has a candidate statement set C. We have to select the right combination of candidate statements among the many target statements $s_{i,j}$ to make the transformed code examples capable of misleading the vulnerability detection model M, i.e., to generate adversarial examples. We can consider the generation of adversarial examples as a game strategy, and the winning condition is successfully making the vulnerability detection model misclassify.

In a search tree, the nodes represent perturbation target statements, and the edges represent the transformations to be performed on perturbation target statements. If the vulnerability detection model predicts incorrectly at a node, then we have found an adversarial example, and the path from the root node to that node represents the full transformations of that example.

As shown in Algorithm 3, first a node object is initialized for each code example as the root of the search tree (line 2). The MCTS search algorithm is primarily divided into four steps (lines 5–10), which are selection, simulation, expansion, and backpropagation. The search is ended by a successful attack (i.e., a model prediction error) or by reaching the maximum number of iterations of the search. After a round of selection and expansion, the best child of the current node is selected as the subsequent root node to continue the search for the best child (line 11). If the attack is successful, the code example of the current node and the path travelled (line 14) are saved. Figure 3d shows the final generated adversarial example, with the red boxes indicating the perturbed statements.

Algorithm 3 Generating adversarial examples

```
Input: $M$ (DL-based vulnerability detector); $p_i$ (vulnerable program); $S_{i,j}$ (the set of statements for perturbation in $p_i$); $x_{i,j}$ (vulnerable code example, i.e., $M(x_{i,j}) = 1$);
Output: An example of adversarial code for an vulnerable program $p_i$
1: $T \leftarrow \varnothing$;
2: $R \leftarrow init(x_{i,j}, S_{i,j})$;{Initialize the root node of the search tree}
3: $r \leftarrow R$;
4: while no success_attack(r) and no stop_iteration(r) do
5:     for $i \leftarrow 1 \ldots 2n$ do
6:         $\nu \leftarrow$ selection$(r, i)$;
7:         $\mu \leftarrow simulation(\nu)$;
8:         $expansion(\nu, \mu, m)$;
9:         $backpropagation(\mu)$;
10:    end for
11:    $r \leftarrow SelecttheBestChildNode(r)$;
12: end while
13: if success_attack(r) then
14:    $Path_r \leftarrow r$;
15:    $exam \leftarrow r$;
16:    $T \leftarrow \{x_{i,j}, exam, Path_r, Success\}$;
17: else
18:    $T \leftarrow \{x_{i,j}, \varnothing, \varnothing, Failed\}$;
19: end if
20: return $T$;
```

Different from the regular MCTS algorithm, we set two parameters here to limit the depth of the algorithm's simulated search and the number of cycles of MCTS. The number of cycles of MCTS is controlled by the length of the candidate statement list n (line 5), which we fix at twice the length of the list. For the depth of the simulation search, we control it by the parameter m (line 8). During the simulation phase, we select three simulation paths at the same time. As shown in Figure 4, we select the one with the highest score as the next node, which reduces the error caused by too much randomness.

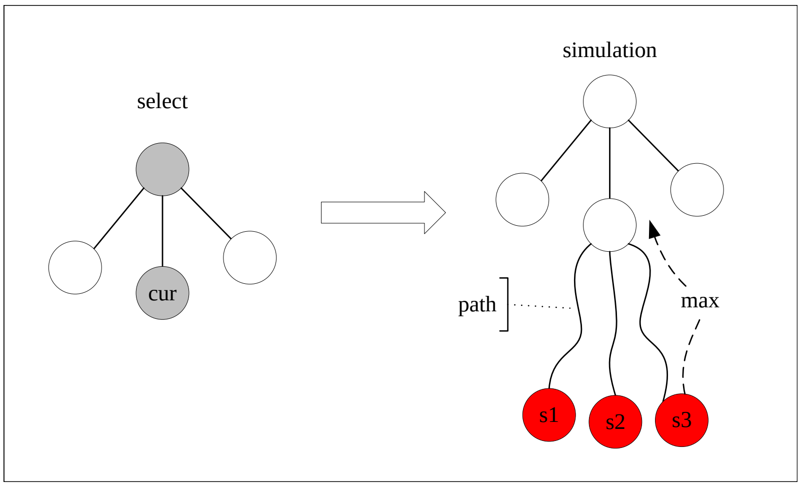


Figure 4. The improved MCTS simulation process.

5. Experiments and Results

5.1. Research Questions

We gear our experiments towards answering the following three Research Questions (RQs):

- RQ1: Is ADVULCODE more effective than existing code adversarial example generation methods?
- RQ2: How do the vulnerability-related lines impact adversarial example generation?
- RQ3: How does the selection of transformation rules impact adversarial example generation?

5.2. Datasets and Evaluation Metrics

For the vulnerability dataset, we need to have a diff file to locate the vulnerability-related lines, and we directly select the SARD and NVD datasets from SySeVR [10].

To evaluate the effectiveness of ADVULCODE in generating source code adversarial examples, we use the widely used comprehensive evaluation metric F-score ($F1$) and the attack success rate metric (ASR). We define TP as predicting positive samples as positive samples, FN as predicting positive samples as negative samples, FP as predicting negative samples as positive samples, and TN as predicting negative samples as negative samples. $F1$ is the harmonic mean of precision and recall, defined formally as:

$$F1 = \frac{2 * precision * recall}{precision + recall}, \tag{1}$$

where precision is calculated by $\frac{TP}{TP+FP}$, and recall is calculated by $\frac{TP}{TP+FN}$. ASR is the number of adversarial examples divided by the total number of examples.

5.3. Implementation

We select three DL-based vulnerability detection models with different combinations of granularity and network: the slice + token + BGRU model SySeVR [10], the function + token + BLSTM model Benchmark [12], and the function + graph + GGNN model Devign [13]. The slice-level model converts each vulnerable code program into many code slices, marks the code slices with labels based on the vulnerable lines, and uses the marked slices as model input. The function-level model splits the vulnerable code program into functions, marks each function in its entirety and uses the marked functions as model input.

We conduct experiments on an Intel (R) Core (TM) i9-9820X machine with a frequency of 3.30 GHz and GeForce RTX 2080.

5.4. Experiments for Answering RQ1

To evaluate the effectiveness of our method, we choose the ALERT [30] method as a baseline. We choose ALERT for the following reasons. Firstly, there are currently no adversarial example generation methods for vulnerability detection models, and, in other tasks of source code processing, many attack methods do not work for vulnerability detection. For example, attacks against authorship attribution models require the dataset to be compilable, whereas vulnerability datasets often comprise vulnerable functions or vulnerability source code that lacks dependencies. Secondly, ALERT is currently the state-of-the-art adversarial example generation method for pre-trained models of code, and ALERT is designed to attack two pre-trained models of code, CodeBERT, and GraphCodeBERT, both of which can be used for vulnerability detection. Therefore, we choose ALERT for comparison with our method.

The experimental results are shown in Table 4, where we observe that our method is better than ALERT. The "-" in the table indicates how much the $F1$ score has decreased compared to the model before the attack. We find that the $F1$ scores of the three models decreased by 22.91%, 12.05%, and 31.14%, respectively, after the attack with our method. In addition, we can observe that, for both the slice + token and function + graph models, the success rate of the ALERT attack is zero. This is because these two vulnerability detection models map all identifier names in the code pre-processing phase. The identifier names are standardized into the same form, such as func_0, func_1, or variable_0, causing the ALERT method ASR to be zero. For the function + token model, however, our method has a 1.10% improvement in attack success rate compared to ALERT. Overall, the ASR of our method is 27.59% higher than ALERT on average.

Insight 1. *Our approach is able to handle models with identifier name mapping and achieve ASR of 16.48% and 65.20%, respectively. Compared to ALERT, our method improves the ASR by an average of 27.59%. Overall, our method is better than ALERT.*

Table 4. Comparing with the existing attack method on the NVD and SARD datasets (unit: %)

Model	**AdVulCode**		**ALERT**	
	F1	***ASR***	***F1***	***ASR***
`Slice + token + BGRU [10]`	60.73 (−22.91)	16.48	83.64 (−0)	0
`Function + token + BLSTM [12]`	84.30 (−12.05)	27.92	83.94 (−12.41)	26.82
`Function + graph + GGNN[13]`	21.66 (−31.14)	65.20	52.80 (−0)	0

5.5. Experiments for Answering RQ2

To further determine the impact of vulnerability-related lines on the model predictions, we consider two generation schemes in the adversarial examples of the vulnerability detection model: transforming only vulnerability-related lines and transforming only non-vulnerability-related lines. We discuss the $F1$ scores and ASR of the model for these two situations to analyze the impact of the vulnerability-related lines on the generation of adversarial examples.

Table 5 illustrates our experimental results, where Vul in the table indicates that only vulnerability-related lines are considered for the transformation attack, and Non-vul indicates that only non-vulnerability-related lines are considered for the transformation attack. We observe that, in these three vulnerability detection models, the $F1$ scores for considering only the vulnerability-related lines are 60.73%, 84.30%, and 21.66%, respectively, which are lower than those for considering only the non-vulnerability-related lines. The ASR of the attacks considering only the vulnerability-related lines is much higher than that of the attacks considering only the non-vulnerability-related lines, which indicates the greater influence of the vulnerable-related lines on the vulnerability detection models.

Insight 2. *The ASR of the attacks considering only the vulnerability-related lines is significantly higher than that of the non-vulnerability-related lines, while the $F1$ scores are 18.47%, 9.63%, and 20.94% lower than those of the non-vulnerability-related lines. Thus, the vulnerability-related lines play a major role in the generation of adversarial examples.*

Table 5. Comparing the impact of vulnerability-related and non-vulnerability-related lines on adversarial example generation (unit: %)

Model	Attack Method	*F*1	*ASR*
`Slice + token + BGRU`	`Vul`	60.73	16.48
	`Non-vul`	79.20	5.17
`Function + token + BLSTM`	`Vul`	84.30	27.92
	`Non-vul`	93.93	8.49
`Function + graph + GGNN`	`Vul`	21.66	65.20
	`Non-vul`	42.60	28.34

5.6. Experiments for Answering RQ3

To evaluate the performance of the attacks after removing the corresponding transformation rules, we count all the rules of adversarial example transformations and select the three most frequently used transformations for elimination experiments. As shown in Figure 5, we find that integer transformation (Rule 10) accounts for the most, followed by pointer-reference transformation (Rule 6) and arithmetic relational operator transformation (Rule 2). We proceed to perform elimination experiments on these three rules to verify their impact on adversarial example generation.

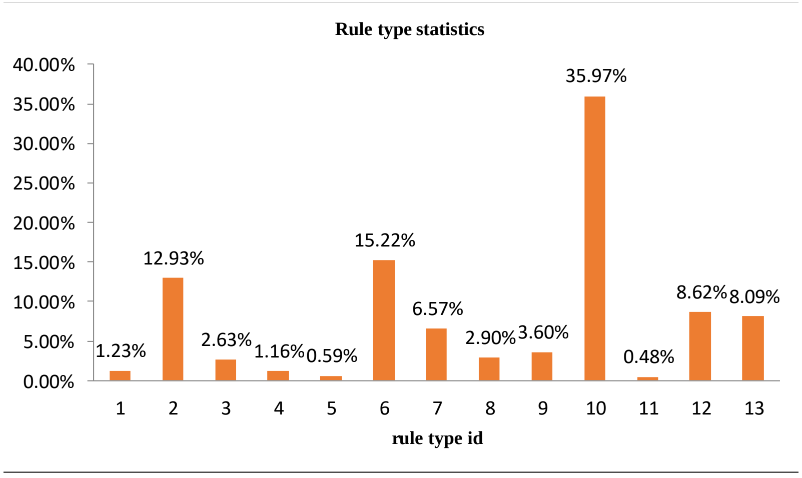


Figure 5. Statistics on the percentage of adversarial example transformation rules.

The results of our experiments are described in Table 6. We observe that, of the three transformation types, integer transformation (Rule 10) accounts for the largest proportion of adversarial examples generated, and also has the largest impact on adversarial example generation. Compared to pointer–reference transformation (Rule 6) and relational operator transformation (Rule 2), the attack success rate for integer transformation is reduced by 9.64%, 16.71%, and 31.58% on the three vulnerability detection models, respectively. We also observe that, in the $Function + token + BLSTM$ model, the $F1$ score after removing these three rules is 90.37%, 91.78%, and 92.88%, respectively, indicating that the difference in the influence of these three rules on this model is not very large. In $Function + graph + GGNN$, the $F1$ score after removing rule 2 is lower than that of rule 6 and rule 10 by 7.38% and

8.82%, respectively. It also indicates that rule 2 is not a major disturbing factor in this model compared with rule 6 and rule 10.

Insight 3. *The transformation rules have different impacts on different models. In short, the relational operator transformation rule has relatively little impact on the adversarial example generation of the Function + graph + GGNN model. Integer transformation is the main factor affecting adversarial example generation.*

Table 6. Comparing the performance of attacks after removing the corresponding transformation rule (unit: %).

Transform Type	*Slice + Token + BGRU*		*Function + Token + BLSTM*		*Function + Graph + GGNN*	
	F1	*ASR*	*F1*	*ASR*	*F1*	*ASR*
`Relational operator transformation (Rule 2)`	66.79	12.73	90.37	17.19	29.94	49.02
`Pointer-reference transformation (Rule 6)`	69.07	10.51	91.78	12.14	37.32	39.03
`Integer transformation (Rule 10)`	77.79	6.84	92.88	11.21	38.16	33.62

6. Limitations

Although our experimental results demonstrate the effectiveness of our method, there are still several limitations in generating adversarial examples. Firstly, the transformation rules in our method are limited by the programming language. We only consider the C and C++ languages; moreover, not every transformation can be applied to all programming languages. Secondly, for the selection of datasets, we selected datasets from SySeVR, where there are relatively few real datasets, and artificial datasets with too much similarity may also reduce the success rate of our attacks. Third, in the graph-based vulnerability detection model, our attack method is time-consuming, which can be attributed to the fact that our method is based on work before data pre-processing. Finally, for preserving vulnerabilities before and after perturbation, we assume that the two programs are semantically equivalent and their vulnerabilities are preserved. While this is reasonable in our setting, there is no guarantee that vulnerabilities will be strictly preserved in all cases. For example, when we transform array definitions, this can also lead to code errors in some cases, such as when memory is exhausted. We acknowledge this limitation, but it does not affect the general validity of our results.

7. Future Work

To the best of our knowledge, the present study is the first to automatically generate adversarial examples by small code transformations for DL-based vulnerability detectors. As can be seen from our limitations, there are ways to make our approach more complete. A follow-up study may be needed in the future, focusing on making the whole framework more compatible with DL-based vulnerability detection models. Firstly, extended programming language support is necessary, and we will consider extending its transformation rules to Java or python languages. Secondly, our graph-based vulnerability detection model is a major drawback in terms of time consumption since our approach performs the transformation before the code is pre-processed. Thus, with the graph-based vulnerability detection model, we regenerate the graph for each transformation, which can be time-consuming. We plan to optimize the graph-based vulnerability detection model attack by performing the transformation of nodes directly on the graph nodes. Finally, in terms of datasets, we used relatively few real datasets in the existing dataset, and we will also consider expanding the real dataset.

8. Related Work

8.1. Prior Studies on Vulnerability Detection for Source Code

Source code vulnerability detection includes code similarity-based [2,3,35–37], rule-based [4,5,38,39], and machine learning-based [6,40–42] methods. The similarity-based method detects vulnerabilities by comparing the similarity of the target code with the vulnerable code. For example, Woo et al. [36] could detect vulnerabilities caused by cloned vulnerable code, but this method could not address vulnerabilities that rely on the C preprocessor. The rule-based approach detects vulnerabilities by matching defined vulnerability patterns. For example, the Flawfind [38] tool used this method to detect vulnerabilities but required experts to manually extract vulnerability features. Machine learning-based methods can be divided into traditional machine learning-based and DL-based. Traditional machine learning-based methods also rely on experts to manually extract vulnerability features. Manual feature extraction is not only time-consuming but is also not easy to fully extract features [9]. DL-based vulnerability detection does not require manually defining vulnerability features, and their features can be divided into the following three categories.

In terms of operational granularity, they can be divided into function [12,13,31] or program slices [9,10,32,43,44]. Lin et al. [13] transformed the entire function into token sequences as model input. Wu et al. [44] processed the source code into fine-grained slices to be embedded in the neural network. In terms of vector representation, it can be used as a vector representation using sequence-based [9,10,32], AST-based [11,33,34], or graph-based [13,45,46] methods. Li et al. [10] transformed source code into sequence form as input, and Zou et al. [13] represented source code as a vector representation of a graph as model input. In terms of neural network structure, there are Bidirectional Gated Recurrent Unit (BGRU) [10], Bidirectional Long Short-Term Memory (BLSTM) [9,33], or Gated Graph Neural Networks (GGNN) [13]), etc. As shown in Table 7, we summarize the target models at which we aimed.

Although DL-based vulnerability detectors have attracted much attention, generating adversarial vulnerable code examples has not been studied until now. In this paper, we design an attack framework that can work on these different vulnerability detection models with different characteristics.

Table 7. Summary of target DL-based vulnerability detection methods.

Model	Granularity	Vector Representation	Network Structure
`SySeVR` [10]	Slice	Token	BGRU
`VulDeePecker` [32]	Slice	Token	BLSTM
`Benchmark` [12]	Function	Token	BLSTM
`Devign` [13]	Function	Graph	GGNN

8.2. Prior Studies on Generating Adversarial Examples

Adversarial example research has shown that perturbing the input can result in misclassification. Currently, there are studies of adversarial examples in image processing [14,15], automatic speech recognition [18–20], natural language processing [22–24], and source code processing [25–28]. However, these adversarial example studies are different from the field of source code processing. Because source code is discrete, it is also limited by strict lexical and syntactic constraints. The study of adversarial examples for vulnerability detection tasks is more challenging. This is because DL-based vulnerability detection is limited in terms of operational granularity and vector representation to add perturbations to the source code.

There is work in the area of source code processing on adversarial sample research for tasks such as code generation, authorship attribution, etc. Zhang et al. [27] treated the selection counter perturbation as a sampling problem and generated adversarial examples by sampling the identifiers. Yang et al. [30] and Yefet et al. [28] used identifier renaming and dead code insertion as code perturbations. Because these methods are based on identifier perturbations, it is difficult for these attack methods to find perturbation points on the statement structure. Quiring et al. [26] generated adversarial examples by changing the entire code style, and this perturbation can lead to an inability to locate the vulnerable lines of the vulnerable code. The ALERT studied by Yang et al. [30] is a state-of-the-art study of adversarial examples for pre-processing tasks, but it has attacked downstream tasks of vulnerability detection, so we use ALERT as a baseline for comparison. As shown in Table 8, we compare our approach with other task methods in terms of identifiers, statement structure, and the operational granularity of the supported DL-based vulnerability detection models.

Our approach to adversarial example generation differs from the above methods in two ways. First, we consider the use of equivalent transformation rules that do not change the semantics of the code, which are small perturbation changes. Second, we consider the presence of vulnerable lines, which also allows us to reduce the overall level of perturbation of the source code while targeting the vulnerable line transformations. Finally, we also improve the MCTS algorithm to guide the generation of adversarial examples.

Table 8. Comparison of adversarial example generation methods.

Method	Identifiers	Statement Structure	Granularity [1]
Zhang et al. [27]	✓	×	Function, Slice
Yang et al. [30]	✓	×	Function, Slice
Quiring et al. [26]	✓	✓	Function
Yefet et al. [28]	✓	×	Function
Our approach	×	✓	Function, Slice

[1] This is operational granularity of DL-based vulnerability detection models.

9. Conclusions

We present the first DL-based adversarial example generation method for vulnerability detection models, adopting an equivalent transformation approach to attack vulnerability detection models in a black-box setting. The method is divided into four main steps: extracting perturbation statements, generating candidate statements, extracting vulnerable-related lines, and replacing candidate statements to generate adversarial examples. For the generation of candidate statements, a strategy is devised to control the degree of perturbation of the candidate statements and the number of generated candidate statements. For the selection of perturbation candidate statements, we use the improved MCTS search algorithm to guide the generation of adversarial examples. The entire attack is treated as a game strategy, with the most effective perturbation added to the target code example. As a result, our attack method can effectively generate adversarial examples. Experimental results demonstrate the effectiveness of the attack method. In addition, the limitations we discuss in Section 6 provide open questions for future research.

Author Contributions: Conceptualization, X.Y. and Z.L.; methodology, X.Y. and Z.L.; validation, X.Y.; formal analysis, X.Y. and Z.L.; investigation, X.Y. and Z.L.; resources, X.Y. and X.H.; data curation, X.Y.; writing—original draft preparation, X.Y.; writing—review and editing, X.Y., Z.L., X.H. and S.Z.; supervision, Z.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Natural Science Foundation of Hebei Province under Grant No. F2020201016. Any opinions, findings, or conclusions expressed in this work are those of the authors and do not reflect the views of the funding agencies in any sense.

Data Availability Statement: The SARD and NVD datasets used to support the findings were derived from the SySeVR article available at https://github.com/SySeVR/SySeVR/tree/master/Program%20data (accessed on 7 March 2022).

Acknowledgments: We thank the reviewers for their constructive comments to help us improve the paper.

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

References

1. Common Vulnerabilities and Exposures. Available online: http://cve.mitre.org/ (accessed on 21 July 2021).
2. Kim, S.; Woo, S.; Lee, H.; Oh, H. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In Proceedings of the 2017 IEEE Symposium on Security and Privacy (S&P), San Jose, CA, USA, 22–24 May 2017; pp. 595–614.
3. 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 (ACSAC), Los Angeles, CA, USA, 5–9 December 2016; pp. 201–213.
4. Checkmarx. Available online: https://www.checkmarx.com/ (accessed on 14 November 2020).
5. HP Fortify. Available online: https://www.ndm.net/sast/hp-fortify (accessed on 19 November 2020).
6. Yamaguchi, F.; Maier, A.; Gascon, H.; Rieck, K. Automatic Inference of Search Patterns for Taint-style Vulnerabilities. In Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P), San Jose, CA, USA, 17–21 May 2015; pp. 797–812.
7. Manès, V.J.M.; Han, H.; Han, C.; Cha, S.K.; Egele, M.; Schwartz, E.J.; Woo, M. The art, science, and engineering of fuzzing: A survey. *IEEE Trans. Softw. Eng.* **2019**, *47*, 2312–2331. [CrossRef]
8. Chen, C.; Cui, B.; Ma, J.; Wu, R.; Guo, J.; Liu, W. A systematic review of fuzzing techniques. *Comput. Secur.* **2018**, *75*, 118–137. [CrossRef]
9. 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. In Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 18–21 February 2018.
10. Li, Z.; Zou, D.; Xu, S.; Jin, H.; Zhu, Y.; Chen, Z. SySeVR: A framework for using deep learning to detect software vulnerabilities. *arXiv* **2018**, arXiv:1807.06756.
11. Lin, G.; Zhang, J.; Luo, W.; Pan, L.; Xiang, Y. POSTER: Vulnerability Discovery with Function Representation Learning from Unlabeled Projects. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS), Dallas, TX, USA, 30 October–3 November 2017; pp. 2539–2541.
12. Lin, G.; Xiao, W.; Zhang, J.; Xiang, Y. Deep Learning-Based Vulnerable Function Detection: A Benchmark. In Proceedings of the 21st International Conference on Information and Communications Security (ICICS), Beijing, China, 15–19 December 2019; pp. 219–232.
13. Zhou, Y.; Liu, S.; Siow, J.; Du, X.; Liu, Y. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS), Vancouver, BC, Canada, 8–14 December 2019; pp. 10197–10207.
14. Yuan, X.; He, P.; Zhu, Q.; Li, X. Adversarial Examples: Attacks and Defenses for Deep Learning. *IEEE Trans. Neural Netw. Learn. Syst.* **2019**, *30*, 2805–2824. [CrossRef] [PubMed]
15. Peng, W.; Liu, R.; Wang, R.; Cheng, T.; Wu, Z.; Cai, L.; Zhou, W. EnsembleFool: A method to generate adversarial examples based on model fusion strategy. *Comput. Secur.* **2021**, *107*, 102317. [CrossRef]
16. Lang, D.; Chen, D.; Huang, J.; Li, S. A Momentum-Based Local Face Adversarial Example Generation Algorithm. *Algorithms* **2022**, *15*, 465. [CrossRef]
17. Lang, D.; Chen, D.; Li, S.; He, Y. An Adversarial Attack Method against Specified Objects Based on Instance Segmentation. *Information* **2022**, *13*, 465. [CrossRef]
18. Qin, Y.; Carlini, N.; Cottrell, G.W.; Goodfellow, I.J.; Raffel, C. Imperceptible, Robust, and Targeted Adversarial Examples for Automatic Speech Recognition. In Proceedings of the 36th International Conference on Machine Learning (ICML), Long Beach, CA, USA, 9–15 June 2019; pp. 5231–5240.
19. Mun, H.; Seo, S.; Son, B.; Yun, J. Black-Box Audio Adversarial Attack Using Particle Swarm Optimization. *IEEE Access* **2022**, *10*, 23532–23544. [CrossRef]
20. Chen, G.; Zhao, Z.; Song, F.; Chen, S.; Fan, L.; Wang, F.; Wang, J. Towards Understanding and Mitigating Audio Adversarial Examples for Speaker Recognition. *arXiv* **2022**, arXiv:2206.03393.
21. Han, S.; Xu, K.; Guo, S.; Yu, M.; Yang, B. Evading Logits-Based Detections to Audio Adversarial Examples by Logits-Traction Attack. *Appl. Sci.* **2022**, *12*, 9388. [CrossRef]
22. Li, J.; Ji, S.; Du, T.; Li, B.; Wang, T. TextBugger: Generating Adversarial Text Against Real-world Applications. In Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 24–29 February 2019.
23. Zhao, T.; Ge, Z.; Hu, H.; Shi, D. MESDeceiver: Efficiently Generating Natural Language Adversarial Examples. In Proceedings of the 2022 International Joint Conference on Neural Networks (IJCNN), Padua, Italy, 18–23 July 2022; pp. 1–8. [CrossRef]
24. Gao, H.; Zhang, H.; Yang, X.; Li, W.; Gao, F.; Wen, Q. Generating natural adversarial examples with universal perturbations for text classification. *Neurocomputing* **2022**, *471*, 175–182. [CrossRef]
25. Rabin, M.R.I.; Wang, K.; Alipour, M.A. Testing Neural Program Analyzers. *arXiv* **2019**, arXiv:1908.10711.
26. Quiring, E.; Maier, A.; Rieck, K. Misleading Authorship Attribution of Source Code using Adversarial Learning. In Proceedings of the 28th USENIX Security Symposium, Santa Clara, CA, USA, 14–19 August 2019; pp. 479–496.
27. Zhang, H.; Li, Z.; Li, G.; Ma, L.; Liu, Y.; Jin, Z. Generating Adversarial Examples for Holding Robustness of Source Code Processing Models. In Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI), New York, NY, USA, 7–12 February 2020.
28. Yefet, N.; Alon, U.; Yahav, E. Adversarial Examples for Models of Code. *arXiv* **2019**, arXiv:1910.07517.

29. Thomas, S.L.; Francillon, A. Backdoors: Definition, Deniability and Detection. In Proceedings of the 21st International Symposium on Research in Attacks, Intrusions, and Defenses (RAID), Heraklion, Greece, 10–12 September 2018; pp. 92–113.
30. Yang, Z.; Shi, J.; He, J.; Lo, D. Natural Attack for Pre-trained Models of Code. *arXiv* **2022**, arXiv:2201.08698.
31. Duan, X.; Wu, J.; Ji, S.; Rui, Z.; Luo, T.; Yang, M.; Wu, Y. VulSniper: Focus Your Attention to Shoot Fine-grained Vulnerabilities. In Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI), Macao, China, 10–16 August 2019; pp. 4665–4671.
32. Zou, D.; Wang, S.; Xu, S.; Li, Z.; Jin, H. *μ*VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Trans. Dependable Sec. Comput.* **2019**, *18*, 2224–2236. [CrossRef]
33. Lin, G.; Zhang, J.; Luo, W.; Pan, L.; Xiang, Y.; de Vel, O.Y.; Montague, P. Cross-Project Transfer Representation Learning for Vulnerable Function Discovery. *IEEE Trans. Ind. Inform.* **2018**, *14*, 3289–3297. [CrossRef]
34. Liu, S.; Lin, G.; Qu, L.; Zhang, J.; De Vel, O.; Montague, P.; Xiang, Y. CD-VulD: Cross-Domain Vulnerability Discovery based on Deep Domain Adaptation. *IEEE Trans. Dependable Sec. Comput.* **2020**, *19*, 438–451. [CrossRef]
35. Jang, J.; Agrawal, A.; Brumley, D. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA, 21–23 May 2012; pp. 48–62.
36. Woo, S.; Hong, H.; Choi, E.; Lee, H. MOVERY: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components. In Proceedings of the 31st USENIX Security Symposium (USENIX Security 22), Boston, MA, USA, 10–12 August 2022; pp. 3037–3053.
37. Bowman, B.; Huang, H.H. VGRAPH: A Robust Vulnerable Code Clone Detection System Using Code Property Triplets. In Proceedings of the 2020 IEEE European Symposium on Security and Privacy (EuroS&P), Genoa, Italy, 7–11 September 2020; pp. 53–69. [CrossRef]
38. Flawfinder. Available online: http://www.dwheeler.com/flawfinder (accessed on 11 October 2020).
39. Gens, D.; Schmitt, S.; Davi, L.; Sadeghi, A. K-Miner: Uncovering Memory Corruption in Linux. In Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 18–21 February 2018.
40. Yamaguchi, F.; Lottmann, M.; Rieck, K. Generalized Vulnerability Extrapolation Using Abstract Syntax Trees. In Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC), Orlando, FL, USA, 3–7 December 2012; pp. 359–368.
41. Neuhaus, S.; Zimmermann, T.; Holler, C.; Zeller, A. Predicting Vulnerable Software Components. In Proceedings of the 2007 ACM Conference on Computer and Communications Security (CCS), Alexandria, VA, USA, 2 November–31 October 2007; pp. 529–540.
42. Grieco, G.; Grinblat, G.L.; Uzal, L.C.; Rawat, S.; Feist, J.; Mounier, L. Toward Large-scale Vulnerability Discovery Using Machine Learning. In Proceedings of the 6th ACM on Conference on Data and Application Security and Privacy (CODASPY), New Orleans, LA, USA, 9–11 March 2016; pp. 85–96.
43. Salimi, S.; Kharrazi, M. VulSlicer: Vulnerability detection through code slicing. *J. Syst. Softw.* **2022**, *193*, 111450. [CrossRef]
44. Wu, T.; Chen, L.; Du, G.; Zhu, C.; Cui, N.; Shi, G. Inductive Vulnerability Detection via Gated Graph Neural Network. In Proceedings of the 2022 IEEE 25th International Conference on Computer Supported Cooperative Work in Design (CSCWD), Hangzhou, China, 4–6 May 2022; pp. 519–524. [CrossRef]
45. Hin, D.; Kan, A.; Chen, H.; Babar, M.A. LineVD: Statement-level Vulnerability Detection using Graph Neural Networks. *arXiv* **2022**, arXiv:2203.05181.
46. Chakraborty, S.; Krishna, R.; Ding, Y.; Ray, B. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Trans. Softw. Eng.* **2022**, *48*, 3280–3296. [CrossRef]