AdVul: Adversarial Attack against ML-based Vulnerability Detection

1st Marina Katoh
School of Cyber Studies
The University of Tulsa
Tulsa, USA
mak5308@utulsa.edu

2nd Weiping Pei
School of Cyber Studies
The University of Tulsa
Tulsa, USA
weiping-pei@utulsa.edu

3rd Youye Xie
The Grainger College of Engineering
University of Illinois Urbana-Champaign
Urbana, USA
youyex2@illinois.edu

***Abstract*—A fundamental problem in software security, detecting software vulnerabilities (i.e. vulnerabilities), has achieved tremendous progress by leveraging machine learning (ML) techniques. These ML-based approaches provide an efficient solution for identifying vulnerabilities and have demonstrated high detection accuracy in recent studies. However, it remains unclear to what extent ML-based vulnerability detection is vulnerable to widespread adversarial attacks. In this paper, we propose a framework that performs Adversarial Attack against ML-based Vulnerability Detection (AdVul) to investigate the vulnerability of the ML-based vulnerability detection. Our experiments on three representative ML models for vulnerability detection demonstrate that the proposed AdVul attack can fool victim models with a high success rate while maintaining the functionality and performance of original programs. Furthermore, we discuss two potential defense methods, enhanced data preprocessing and adversarial training. Our evaluation demonstrates that both methods can to a meaningful extent mitigate the proposed AdVul adversarial attacks. More broadly, our work intends to prompt the security community to seriously consider the potential risks associated with the extensive use of ML techniques in the security domain.**

***Index Terms*—adversarial examples, software vulnerability detection, machine learning**

1. Introduction

With the ubiquitous usage of software in modern life, software vulnerabilities have become a critical concern. Software vulnerabilities are defined as "specific flaws or oversights in a piece of software that allow attackers to do something malicious: expose or alter sensitive information, disrupt or destroy a system, or take control of a computer system or program" [1]. Given the potential severe consequences, significant efforts have been made over the past few decades to mitigate the risks posed by software vulnerabilities. Among these efforts, machine learning (ML) techniques have shown remarkable success in detecting vulnerabilities [2], [3].

However, recent studies have demonstrated that ML models are vulnerable to adversarial attacks [4]–[6]. Specifically, an adversary could force a well-trained ML model to make incorrect predictions by intentionally crafting adversarial examples. This involves introducing small perturbations to the original examples to generate adversarial examples that will be misclassified by the model while preserving the original semantics. For instance, in image classification, imperceptible noise can be added to an image to change the model's prediction [7], [8]. Similarly, in sentiment analysis, a word can be replaced with its synonym in a sentence to change the model's prediction [9], [10]. Studying adversarial example generation not only reveals the vulnerability of existing models but also helps researchers explore better defense solutions to enhance the robustness of models. For example, adversarial training has been proven to be one of the most effective defense schemes [11], [12].

Generating adversarial examples in programs differs from generating adversarial images in computer vision, as it involves handling discrete data, much like in natural language processing (NLP). However, adversarial example generation in program is even more challenging than in NLP for two main reasons. First, adversarial programs must adhere to strict syntactic constraints to remain both compilable and executable. Programming languages have rigorous syntax rules to ensure the unambiguous interpretation of code, whereas natural languages have more flexible syntax. For example, character-level perturbations, such as introducing typos, are commonly used to generate adversarial examples in NLP tasks [13], [14]. However, similar perturbations in code, such as typos or mismatched variable names, result in compilation errors and bugs, rendering the program non-executable. Therefore, approaches effective in NLP are not directly transferable to the domain of programs. Second, even minor executable perturbations in code can easily alter the program's original functionality. As a result, the space of feasible perturbations for generating adversarial examples in programs is significantly smaller than that in NLP, making adversarial examples generation in programs far more challenging.

While existing research primarily focus on investigating the feasibility of using ML models to detect vulnerabilities [2], [3], [15], few attention has been given to the vulnerabilities introduced by the ML models themselves. To fill this gap, we investigate the vulnerability of ML-based vulnerability detection to adversarial attacks by measuring

the extent to which attackers can evade detection by generating adversarial examples and the extent to which non-vulnerable programs could be misclassified as vulnerable. To this end, we propose a framework named AdVul, which adopts a greedy based approach for perturbation selection to perform adversarial attacks against ML-based vulnerability detection. To facilitate the generation of adversarial examples in programs, we introduce a set of functionality preserving transformations. We then evaluate the effectiveness of our framework by conducting experiments on popular and important ML models for vulnerability detection. Our experimental results show that AdVul achieves success rates of 42.00%, 44.50%, and 67.50% on the LineVul model [3], the Vuldeepecker model [2], and the GNN-ReGVD Model [15], respectively. We make our code publicly available in [16].

In this paper, we make five main contributions:

- We introduce a set of functionality-preserving transformations and assess the robustness of state-of-the-art ML models against each type of transformation.
- We propose a framework, AdVul, with two stages, important code snippet selection and greedy based perturbation, to generate adversarial examples in programs against ML-based vulnerability detection.
- We demonstrate the effectiveness of AdVul by conducting experiments on three representative ML models for vulnerability detection.
- We explore and evaluate two potential defense methods, enhanced data preprocessing and adversarial training. Results demonstrate that both methods can to a meaningful extent mitigate adversarial attacks.
- More broadly, our work intends to prompt the security community to seriously consider the potential risks posed by the widespread use of machine learning techniques in the security domain.

2. Related Work

2.1. ML-based Vulnerability Detection

In recent years, there has been extensive research on ML-based vulnerability detection. The methods mainly fall into three categories based on the granularity of detection: file/function/method level, code gadget level, and line level. For example, Nguyen at al. proposed GNN-ReGVD model, which treats source code as a flat sequence of tokens and utilizes a Graph Neural Network (GNN) to learn inherent structure of source code for detecting vulnerabilities at the function level [15]. To address the need for finer-grained detection, Li et al. introduced the concept of "code gadgets," which refers to a set of (not necessarily consecutive) semantically related lines of code. They proposed the Vuldeepecker model, using Bidirectional Long Short-Term Memory (BLSTM) neural network to detect vulnerabilities at the code badget level [2]. Similarly, Li et al. [17] developed the IVDetect model, which leverages a Feature-attention Graph Convolution Network (FA-GCN) approach to predict function-level vulnerabilities and uses the GNNExplainer to locate the fine-grained location of vulnerabilities. For even finer granularity, Fu and Tantithamthavorn proposed the LineVul model, a Transformer-based approach designed for line-level vulnerability detection [3]. Their experimental results showed that LineVul outperformed other models, including IVDetect approach, in detecting vulnerabilities with higher accuracy. All these studies have demonstrated the impressive effectiveness of using ML techniques to detect vulnerabilities. However, little is known about the robustness of existing models against adversarial attacks. Our work aim to investigate the feasibility of adversarial attacks on ML-based vulnerability detection. We evaluate our proposed attack framework on three representative models at different levels of detection granularity.

2.2. Adversarial Attacks in Program Analysis

ML models have shown astonishing results in many domains but have been found to be vulnerable to adversarial attacks, which create adversarial examples to intentionally force a trained ML model to make incorrect predictions. Some research studies have investigated the adversarial attacks in program analysis. For example, Zhou et al. proposed ACCENT, an identifier substitution approach to craft adversarial code snippets in the code comment generation task [18]. Yefet et al. proposed Discrete Adversarial Manipulation of Programs (DAMP), which applies two semantic-preserving transformations, variable renaming and dead-code insertion, to generate adversarial examples [19]. They conducted evaluation on the code summarization task [20] and the variable misuse detection task [21]. Chen et al. proposed a set of program transformations involving identifier renaming and structural transformations and perform adversarial attacks [22]. These studies mainly performed adversarial attacks in a white-box setting, where adversaries have full access to either the model under attack or to a similar model. In contrast, we focus on attacking ML-based vulnerability detection in a black-box setting, which is more practical and realistic. In this setting, attacks are performed without any knowledge about the ML models. The only information provided by the model is whether the input source code is vulnerable, along with a confidence score.

3. Problem Formulation and Threat Model

3.1. Problem Formulation

In ML-based vulnerability detection, the inputs are typically the source code programs. We use x to represent a program input that consists of a sequence of lines of code, denoted by $x = (l_1, ..., l_n)$. A trained model can be represented as F: $X \rightarrow Y$, which maps the program space X to two classes, $Y = \{0, 1\}$, where 0 represents non-vulnerable and 1 represents vulnerable. For an original program $x_{orig} \in X$ with label $y_{orig} = F(x_{orig})$, our attack aim to generate an adversarial example x_{adv} such that the predicted label $y_{adv} = F(x_{adv})$ differs from y_{orig}. An adversarial example

is typically generated by introducing perturbations Δx to the original example: $x_{adv} = x_{orig} + \Delta x$. The generation of x_{adv} needs to satisfy three criteria: (1) x_{adv} should be compliable and executable, (2) x_{adv} preserves the same functionality as that in x_{orig}, and (3) the trained victim model F predicts different labels for x_{orig} and x_{adv}. The goal of the adversarial attack can be deviating the label to incorrect one (i.e., $y_{adv} \neq y_{orig}$) which is known as *untargeted* attack, or specified one ($y_{adv} = y_{target}$) which is known as *targeted* attack. In the context of vulnerability detection, which is a binary classification task, untargeted attacks and target attacks are effectively equivalent. Thus, we set $y_{target} = \sim y_{orig}$.

3.2. Threat Model

In general, adversarial attacks can be performed in three settings: black-box, white-box, and grey-box. The adversarial attacks we formulated can be performed in all these settings, but for this work, we focus specifically on performing and evaluating adversarial attacks in the black-box setting. More specifically, the adversary does *not* have access to training data, learning algorithms, model architectures, parameters, or model gradients. It can only query a target or victim model with input programs and get at most the prediction results along with confidence scores. We focus on the black-box setting for two major reasons. First, the black-box setting provides the adversary with the most limited capabilities, thus it is often considered as the most practical setting for attackers. It is most popularly considered in the computer vision, NLP, and computer security domains [23]–[26]. Second, due to the limited capabilities granted to the adversary, it is more challenging to perform successful adversarial attacks in the black-box setting compared to white-box or grey-box settings. While our focus is on black-box adversarial attacks, extending our approach to white-box and grey-box settings can be straightforward.

4. Generating Adversarial Examples for ML-based Vulnerability Detection

Figure 1 illustrates our proposed AdVul framework for performing adversarial attacks against ML-based vulnerability detection. The framework consists of two stages: (1) important code snippet selection and (2) greedy-based perturbation. In the first stage, we segment the program into multiple snippets using a sliding window and measure the importance of each snippet in relation to the model's prediction. In the second stage, we generate a set of functionality-preserving perturbations for the most important snippets. These candidate perturbations are then applied using a greedy approach to maximize the attack's success while minimize the number of perturbations introduced.

4.1. Important Code Snippet Selection

To minimize the number of modifications and maintain the similarity between the original and the perturbed code as much as possible, we propose to apply perturbations to important parts of the code that significantly influence the model prediction. This region of interest ranking and selection mechanism has been successfully used in prior adversarial examples generation studies in computer vision [27] and NLP [14], [28]. However, unlike NLP tasks, where perturbations can be applied at the *token* level (e.g., through word replacement), token level perturbations in a program, such as identifier or operator replacement, can lead to unanticipated changes in the functionality or even render the program non-executable. While one may suggest parsing the program at the *function* level, it is often too coarse-grained, as in most cases, only a few specific lines of code within a function contribute to the vulnerability. Therefore, we propose to parse the program at the *snippet* level, where each snippet consists of multiple lines of code, and select the important snippets for perturbation.

Step 1: Parse the source program. We first segment the program into individual lines. The lines are separated by either a newline marker or a statement terminator, such as a semicolon (;) in languages like C/C++. We denote the input program $x = (l_1, l_2, ..., l_n)$, where l_i represents the i^{th} line of the program. Then, given a window of size w, we refer a code snippet s_i consisting of w consecutive lines of code as $s_i = (l_i, l_{i+1}, ..., l_{i+w-1})$. A program with n lines of code will generate $(n - w + 1)$ code snippets in total.

Step 2: Calculate the importance of the snippets. To measure the influence of a code snippet s_i towards the model's prediction score $F_y(x)$ for the label y, we calculate the difference between the prediction scores before and after removing the snippet. We use $x_{-i} = (l_1, ..., l_{i-1}, l_{i+w}, ..., l_n)$ to represent the input program with the snippet s_i removed. Previous research on black-box adversarial attacks in NLP has focused on finding important words/sentences whose removal reduces the prediction scores using $F_y(x) - F_y(x_{-i})$ as the importance score, such as the Eq.3 in [14] and Eq.2 in [28]. Different from their approaches, we propose to consider not only snippets that decrease the prediction confidence but also those that increase it. This is achieved by measuring the absolute value of the difference in model predictions before and after removing the snippet, capturing both positive and negative influence on the model's decision. We apply the following scoring function to determine the importance of the i-th snippet in a program x:

$$C_i = \begin{cases} |F_y(x) - F_y(x_{-i})|, \text{if } F(x) = F(x_{-i}) = y \\ |F_y(x) - F_y(x_{-i})| + |F_{\bar{y}}(x) - F_{\bar{y}}(x_{-i})|, \\ \text{if } F(x) = y, F(x_{-i}) = \bar{y}, \text{and } y \neq \bar{y} \end{cases} \quad (1)$$

where $F(\cdot)$ is the predicted label, and $F_y(\cdot)$ denotes the prediction score for the label y. Similar to other scoring functions in the literature [14], [28], our proposed scoring function reflects the importance of snippets based on their influence on the prediction. This method is suitable for black-box attacks since it does not require access to the model's parameters, loss function, or structure. We will compare the effectiveness of our scoring function with the one proposed in [28] in Section 5.3

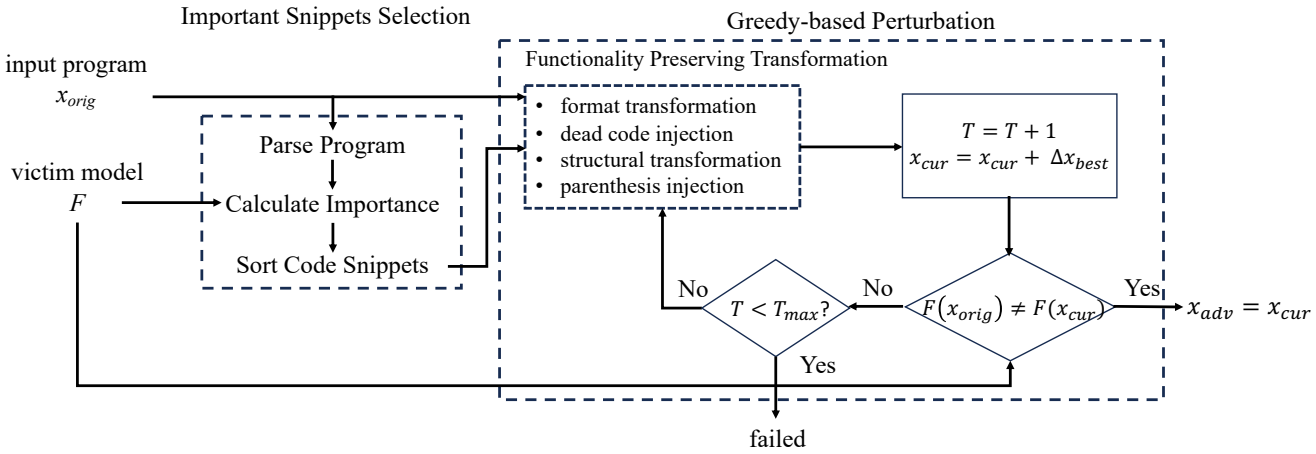


Figure 1: Our AdVul framework for generating adversarial examples.

Step 3: Sort and select candidate snippets. Once the importance score for each snippet is calculated using Eq. 1, we sort the snippets in descending order based on their scores. The top k snippets are then selected to form the candidate snippet set $S_{candidate}$, which will be used for perturbation.

4.2. Greedy-based Perturbation

After identifying the important snippets in the first stage, the second stage begin by applying functionality-preserving transformations to each snippet to generate candidate perturbations (Section 4.2.1). A greedy search is then performed to select the most effective perturbation (Section 4.2.2).

4.2.1. Functionality Preserving Transformation. For a given snippet s_i with a high importance score, as obtained via Eq. 1, perturbations must follow the syntax rules of the programming language and preserve the original program's functionality. To achieve this, we propose a set of functionally equivalent transformations, which can be classified into four categories: (1) format transformation, (2) dead code insertion, (3) structural transformation, and (4) parenthesis insertion. Since most existing ML-based vulnerability detection datasets are based on C/C++ programming languages, the transformations are detailed based on C/C++ syntax.

Format transformation. In C/C++ programs, whitespace such as spaces, tabs, and newlines are used for formatting purposes. Adding or removing these whitespace will not incur any syntax errors or change the functionality of the code. We implement four types of format transformations in this work: 1) adding an empty line, which inserts a blank line between existing lines of code; 2) removing an empty line, which identifies and deletes a randomly selected blank line from the program; 3) adding a tab, which inserts a tab between lines or at the beginning of a line; and 4) removing a tab, which detects and randomly removes a tab from an indented line.

Dead code insertion. A dead code refers to a code that do not affect the program's functionality, such as comments or unused variables. This makes dead code insertion an ideal functionality-preserving transformation for adversarial perturbation, as the dead code is imperceptible from the perspective of program execution. In this work, we insert the single variable initialization as the dead code. Specifically, we create a new (unused) variable, assign it a random integer value, and insert this dead code at random locations between statements in the important snippet. For example, a new variable "*int newVar = 66;*" could be added without affecting the program's logic. While detecting and removing unused variables before feeding the program to ML models could theoretically make this type of perturbation invalid, such preprocessing is rarely enforced by existing vulnerability detection models. Even if applied, adversaries could easily circumvent it by considering more complex forms of dead code, such as "*bool dummy = true; if (dummy){ ... }*", which are harder for automated tools to detect.

Structural transformation. Structural transformations involve modifying the structure of the code while preserving its original functionality. We focus on three types of structural transformations. The first one is comparison operator exchange, which involves extracting and swapping the operands of the comparison operations and adjusting the operator accordingly. For example, the expression "$a > b$" is functionally equivalent to "$b < a$", and "$a == b$" is equivalent to "$b == a$". The second one is increment and decrement operator exchange. Pre-increment and post-increment are equivalent in cases where increment and decrement operators are used in a single expression. For example, "$i + +$;" is equivalent to "$+ + i$;" when used in isolation, as both modify the variables by 1 and yield the same result. The third one is compound assignment operator exchange. In C/C++ programs, compound assignment operators such as "$+ =$" are shortcuts that perform a math operation and assignment in a single step. These can be expanded back to basic assignment expressions. For

example, "$a+=b$" can be converted to "$a=a+b$" without changing the original functionality. All these transformations alter the structure of the code without changing the logic or behavior of the program.

Parentheses insertion. We also consider parentheses insertion, which are commonly used to control the order of expression evaluation in C/C++ programs. Instead of altering the operator precedence in the original programs, we propose to insert parentheses to group expressions without change their logic or behaviors. For example, the expression "$a=b+c;$" can be transformed into "$a=(b+c);$".

We utilize the four types of functionality-preserving transformations described above to perform perturbations on the selected snippets. While it is impractical to exhaustively enumerate all possible functionality-preserving transformations, we focus on representative use cases for each type of transformation. For example, in the case of dead code insertion, we apply single variable initialization as a representative example in this work.

4.2.2. Greedy Search for Perturbation. With a set of candidate transformations, we apply greedy search approach to select the most effective perturbation. It takes as input the current program x_{cur}, which could either be the original program x_{orig} or a perturbed program. To choose the best transformation, we follow these steps:

- Step 1: We obtain all possible transformations for the top k important code snippets and generate candidate programs by applying each possible transformation to the current program x_{cur}: $x_{cur}+\Delta x_0, x_{cur}+\Delta x_1, ..., x_{cur}+\Delta x_n$
- Step 2: We calculate the prediction scores of all perturbed programs and pick the transformation that has the maximum prediction score of the target label y_{target}.
- Step 3: The selected transformation is applied to the current program $x_{cur}=x_{cur}+\Delta x_{best}$. If the victim model predicts the label for this perturbed program different from that for the original program $F(x_{orig}) \neq F(x_{cur})$, the current program is returned as a successful adversarial example. Otherwise, we repeat steps 1 to 3 until we either find a successful perturbation or reach a predefined maximum number of attempts T_{max}.

By applying this greedy selection strategy, we can generate adversarial examples with minimal perturbations.

4.3. High-level Algorithm

Algorithm 1 outlines the entire procedure of our proposed adversarial attack against ML-based vulnerability detection. It takes as input the original program x_{orig} , the victim model F, and a hyper-parameter T_{max}, representing the maximum number of perturbation iterations. The output is either a successful adversarial program x_{adv} or "None" if the attack is unsuccessful. The algorithm begins by identifying important code snippets (from line 1 to line 7). Following this, a greedy-based approach is used to iteratively apply perturbations to the program (from line 8 to line 19). In each iteration, functionality-preserving transformations are applied to the important snippets, and the transformation that results in the greatest deviation from the original program's prediction is selected for perturbation. By limiting changes to the best transformation Δx_{top}, the program's functionality is preserved, while the modifications remain subtle and largely imperceptible by users. Regarding the hyper-parameter T_{max}, our experiments show that most of the generated programs do not change after about 10 iterations, indicating that increasing this value beyond 10 does not significantly improve the attack's success rate. Therefore, we set $T_{max}=10$.

Algorithm 1 Adversarial Attack against ML-based Vulnerability Detection

Inputs: an original program x_{orig}, a victim model F, window size w, number of snippets k, and the maximum iterations T_{max}
Output: an adversarial program x_{adv} or None.

Important snippets selection
1: $x_{orig}=(l_1, l_2, ..., l_n)$
2: **for** $i=1, 2, ..., n-w$ **do**
3: $s_i \leftarrow (l_i, l_{i+1}, ..., l_{i+w})$
4: $c_i \leftarrow$ compute the importance score using Eq. 1.
5: $S.append(s_i), C.append(c_i)$
6: **end for**
7: `sort(`S`, key=`C`)`
Greedy-based perturbation
8: $x_{cur} \leftarrow x_{orig},\ i \leftarrow 0$
9: **while** $i++ < T_{max}$ **do**
10: **for** `s in S` **do**
11: $x_{pert} \leftarrow x_{cur} + \Delta(s, \text{TRANSFORM}(s))$
12: **if** $F(x_{pert}) \neq F(x_{orig})$ **then return** x_{pert}
13: **else**
14: $X_{pert}.append(x_{pert})$
15: **end if**
16: **end for**
17: `sort(`X_{pert}`)`
18: $x_{cur} \leftarrow X_{pert}[0]$
19: **end while**
20: **return** None

5. Evaluation

In this section, we evaluate our proposed AdVul adversarial attack on three ML models for vulnerability detection.

5.1. Targeted Models and Datasets

The victim models we use in this paper are the LineVul model [3], the VulDeePecker model [2], and the GNN-ReGVD model [15]. Among these, LineVul is considered

the state-of-the-art in vulnerability detection, having demonstrated outperformance compared to other models. We evaluate those models on datasets that used in prior research studies. Specifically, we evaluate the LineVul model on the Fan dataset [29]. This dataset is collected from 348 open-source Github projects, covering 91 different Common Weakness Enumerations (CWEs) from 2002 to 2019. It includes 188,636 C/C++ functions with 5.7% labeled as vulnerable. We evaluate the VulDeePecker model on the CWE-399 dataset, which is derived from National Vulnerability Database (NVD) and contains 7,285 code gadgets correspond to resource management error vulnerabilities. We evaluate the GNN-ReGVD model on the CodeXGLUE dataset, which is a benchmark constructed by Microsoft and aggregates 14 datasets from previous studies.

5.2. Effectiveness of Transformations

We begin by assessing each type of functionality-preserving transformation introduced in Section 4.2.1. Since each type of transformation can be implemented in multiple ways (e.g., there are multiple possible locations for inserting an empty line within the program), we conduct two experiments to evaluate their effectiveness. These experiments represent two extreme scenarios: (1) applying a single perturbation, where we apply just one perturbation from a given transformation type, representing the lower bound of potential modifications; and (2) applying all possible perturbations, where we apply every possible perturbation from a single transformation type, providing an upper bound of potential modification.

5.2.1. Effectiveness of Applying A Single Perturbation. For each type of transformation, we identified all possible perturbations and then applied one randomly selected perturbation. The overall results of applying a single perturbation from different types of transformations are listed in Table 1. The results show that even a single perturbation can decreases the accuracy of the victim models to some extent, though the impact varies by transformation type. For the LineVul model, dead code insertion has the most significant impact with 8.92% decrease to the accuracy, while single perturbations from other transformation types cause only slight accuracy drops, generally less than 1.00%. Similarly, for the Vuldeepecker model, adding one single line of dead code results in the largest accuracy decreases, at 6.34%. However, the GNN-ReGVD model is more robust against single perturbations, with the maximum accuracy drop being less than 0.3%. These results indicate that different models exhibit varying levels of robustness against individual perturbations from our proposed functionality-preserving transformations.

5.2.2. Effectiveness of Applying All Perturbations from A Single Type of Transformation. For each type of transformation, we identified and applied all possible perturbations. The overall results are listed in Table 2. We can see that in some cases, applying all possible perturbations can further decrease the accuracy of the victim models compared to a single perturbation (Table 1). For example, removing a single tab decreases the accuracy of the LineVul model to 98.63%, while removing all tabs decreases the accuracy to 8.67%. This drastic decrease underscores the cumulative effect of multiple perturbations. Similarly, for the Vuldeepecker model, adding a single line of dead code decreases the accuracy to 84.49%, while adding multiple lines of dead code further decreases the accuracy to 66.51%. This implies that the introduction of more dead code generate more noises, which further distracts the victim model from correctly identify vulnerabilities. These results indicate varying impacts of different perturbations and suggest the cumulative effects of multiple perturbations when performing adversarial attacks.

The results from Table 1 and Table 2 demonstrate that our proposed functionality-preserving transformations can to some extent decrease the accuracy of the ML-based vulnerability detection, implying the feasibility of applying those transformations as perturbations for adversarial attacks. However, we also observed that different models exhibit varying levels of robustness against different types of transformations. It highlights the challenges of conducting successful adversarial attacks across different ML models.

5.3. Effectiveness of Greedy-based AdVul

Instead of limiting perturbations to a single transformation type, our proposed AdVul framework considers perturbations from various transformation types to conduct more effective adversarial attacks. It identifies all possible perturbations across all transformation types for the top k most important snippets, then uses a greedy-based approach to select the most effective perturbation. This strategy results in an optimal combination of perturbations, aiming to mislead the victim model while minimizing the number of perturbations applied.

To evaluate the effectiveness of our framework, we first identify valid examples that are correctly predicted by the victim models. From these, we sample 100 valid positive examples (i.e., vulnerable programs) and 100 valid negative examples (i.e. non-vulnerable programs) as the original programs to perform our proposed AdVul attack. We evaluate our framework by examining the quality of generated examples in terms of ***success rate***, which is the percentage of adversarial examples that successfully mislead the victim model into making incorrect predictions. Additionally, to assess how well we preserve the content of the original program, we measure the ***average number of perturbations*** added to original examples.

We evaluate our proposed AdVul attack against three victim models, the LineVul model, the Vuldeepecker model, and the GNN-ReGVD model. Experiments are conducted using with different parameters, including the maximum number of iteration T_{max}, the window size w for parsing the program, and the number of selected snippets k. We also compare three different scoring functions for snippets selection: (1) randomly selecting k snippets; (2) sorting and

TABLE 1: Accuracy of ML Models for Vulnerability Detection with A Single Perturbation.

Model	Dataset	Original Accuracy	Performance after Perturbation							
			Add Empty Line	Remove Empty Line	Add Dead Code	Add Parentheses	Comp Operator Exchange	Other Operator Exchange	Add Tab	Remove Tab
LineVul [3]	Fan	99.11%	99.09%	99.1%	90.19%	99.13%	98.29%	99.12%	99.03%	98.63 %
Vuldeepecker [2]	CWE-399	90.83%	90.83%	90.83%	84.49%	90.80%	86.72%	90.84%	89.29%	90.80 %
GNN-ReGVD [15]	CodeXGLUE	59.22%	59.22%	59.22%	59.19%	58.97%	59.11%	59.15%	59.22%	59.22 %

TABLE 2: Accuracy of ML Models for Vulnerability Detection with All Perturbations from A Single Type of Transformation

Model	Dataset	Original Accuracy	Performance after Perturbation							
			Add Empty Line	Remove Empty Line	Add Dead Code	Add Parentheses	Comp Operator Exchange	Other Operator Exchange	Add Tab	Remove Tab
LineVul [3]	Fan	99.09%	99.06%	99.09%	85.85%	99.1%	98.01%	99.08%	98.92%	8.67 %
Vuldeepecker [2]	CWE-399	90.83%	90.80%	90.80%	66.51%	90.12%	86.68%	90.84%	85.42%	90.80 %
GNN-ReGVD [15]	CodeXGLUE	59.22%	59.22%	59.22%	59.19%	58.16%	59.04%	59.19%	59.22%	59.22 %

selecting top k snippets based on the scoring function proposed by [28]; and (3) sorting and selecting top k snippets using the scoring function defined in Eq. 1. The detailed evaluation results of the AdVul attack on three models are presented in Table 3, Table 4, and Table 5, respectively.

Overall Results and Analysis. Our proposed AdVul attack achieves a success rate of up to 42.00% on the LineVul model, 44.50% on the Vuldeepecker model, and 67.50%on the GNN-ReGVD model with fewer than 2 perturbations on average (as indicated by the "average # perts" in the tables). These results demonstrate that our framework effectively preserves most of content and functionality of the original programs while successfully misleading the models. We will provide a detailed analysis of how how various settings and hyper-parameter affects the success rate of the AdVul attack.

Maximum Number of Iteration T_{max}. We first investigate how the maximum number of iterations T_{max} affects the final results. Specifically, we conduct experiments with $T_{max} = 10$, 15, 20 when attacking the LineVul model. The results listed in Table 6 indicate that increasing the maximum number of iterations only boosts up the success rate by a small margin. Furthermore, when the number of selected snippets is relatively small (e.g., $k = 5$ and $k = 10$), the success rate remains the same for $T_{max} = 10$, 15, 20. This result implies that the proposed AdVul attack is efficient and effective, only requiring a few iterations to perform successful attacks. Consequently, we will set $T_{max} = 10$ for all subsequent experiments.

Window Size w. When attacking the LineVul model (Table 3), the AdVul framework achieves the best success rate of 38.5% when window size $w = 3$ and 42.00% with $w = 5$. Using the same snippet selection mechanism and the same number of selected snippets, the success rate with $w = 5$ is higher than that with $w = 3$. This result is intuitive, as a larger window size is more likely to capture important semantic relations within a snippet, allowing the AdVul attack to target and disrupt these relationships more effectively. However, using a larger window size also demands more computational resources and increase the possibility of destroying the program's original functionality. Similarly, when attacking the GNN-ReGVD model (Table 5), the success rate with $w = 5$ (67.50%) is higher than that with $w = 3$ (63.00%). Interestingly, this pattern did not hold for the Vuldeepecker model, where the smaller window size of $w = 3$ yields a higher success rate (44.50%) compared to $w = 5$ (41.50%).

Number of Snippets k. For all three victim models, increasing the number of selected snippets for perturbation leads to a higher success rate. This is expected, as selecting more snippets provides more opportunities for perturbations, thereby increasing the likelihood of generating successful adversarial examples. However, this also results in a higher average number of perturbations, indicating more modifications are involved. Similar to effect of the window size, while more perturbations can improve attack success, they also introduce greater computational overhead and increase the risk of discrupting the program's original functionality.

Snippet Selection Mechanism. In comparing different mechanisms for snippet selection, we found that our proposed scoring function is on par with the mechanism presented in [28] when $w = 3$, while it outperforms the existing scoring function when $w = 5$ for the LineVul model. Although our proposed scoring function does not outperform the other two functions, it still achieves comparable results for the Vuldeepecker model. As for the GNN-ReGVD model, our proposed scoring function achieves the highest success rate with $w = 3$, while random snippet selection yields the best result with $w = 5$. These results indicate that it is beneficial to rank and consider snippets that both reduce prediction confidence and those that enhance it for adversarial attacks.

Distribution of Successful Adversarial Examples. In addition to the success rate and the average number of perturbations, Tables 3, 4, and 5 show the distribution of successful adversarial examples on positive and negative examples. The column named "p2n" lists the number of adversarial examples that are generated based on positive examples (i.e., vulnerable programs) and misclassified as negative (i.e., non-vulnerable) by the victim model. The success of these examples indicates that attackers are able to evade ML-based on vulnerability detection. The column named "n2p" lists the number of the adversarial exam-

TABLE 3: Evaluation Results on Attacking LineVul model ($T_{max} = 10$).

	window size w= 3					window size w= 5				
Snippet Selection	top k snippets	success rate	average # perts	p2n	n2p	top k snippets	success rate	average # perts	p2n	n2p
random	5	19.50%	1.33	2	37	5	32.50%	1.54	3	62
	10	28.00%	1.52	2	54	10	35.00%	1.67	4	66
	15	31.50%	1.7	4	59	15	39.50%	1.78	5	74
	20	34.50%	1.77	5	64	20	39.50%	1.67	5	74
Eq. used in [28]	5	21.00%	1.31	4	38	5	31.00%	1.74	5	57
	10	31.00%	1.89	7	55	10	37.00%	1.77	7	67
	15	37.50%	1.75	8	67	15	37.50%	1.75	8	67
	20	38.50%	1.73	8	69	20	38.50%	1.73	8	69
Our proposed Eq. 1	5	22.50%	1.38	4	41	5	33.50%	1.6	5	62
	10	32.50%	1.71	7	58	10	39.50%	1.86	7	72
	15	35.50%	1.77	8	63	15	42.00%	1.88	8	76
	20	38.00%	1.82	8	68	20	42.00%	1.83	8	76

TABLE 4: Evaluation Results on Attacking Vuldeepecker Model ($T_{max} = 10$).

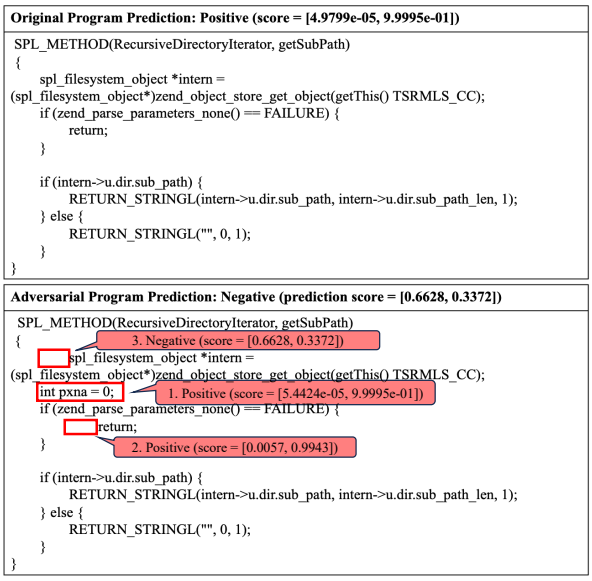
	window size w= 3					window size w= 5				
Snippet Selection	top k snippets	success rate	average # perts	p2n	n2p	top k snippets	success rate	average # perts	p2n	n2p
random	5	28.50%	1.32	30	27	5	28.00%	1.32	34	22
	10	37.00%	1.32	44	30	10	37.00%	1.44	50	24
	15	43.00%	1.43	54	32	15	41.00%	1.44	57	25
	20	44.50%	1.42	57	32	20	41.50%	1.40	58	25
Eq. used in [28]	5	27.50%	1.24	27	28	5	26.00%	1.42	30	22
	10	35.00%	1.53	40	30	10	29.50%	1.32	35	24
	15	39.00%	1.55	48	30	15	39.00%	1.47	56	22
	20	44.50%	1.42	57	32	20	41.00%	1.39	57	25
Our proposed Eq. 1	5	25.50%	1.27	23	28	5	29.50%	1.59	37	22
	10	33.50%	1.63	38	29	10	32.00%	1.48	41	23
	15	37.50%	1.57	44	31	15	37.50%	1.45	52	23
	20	43.50%	1.44	56	31	20	41.00%	1.41	58	24

TABLE 5: Evaluation Results on Attacking GNN-ReGVD Model ($T_{max} = 10$).

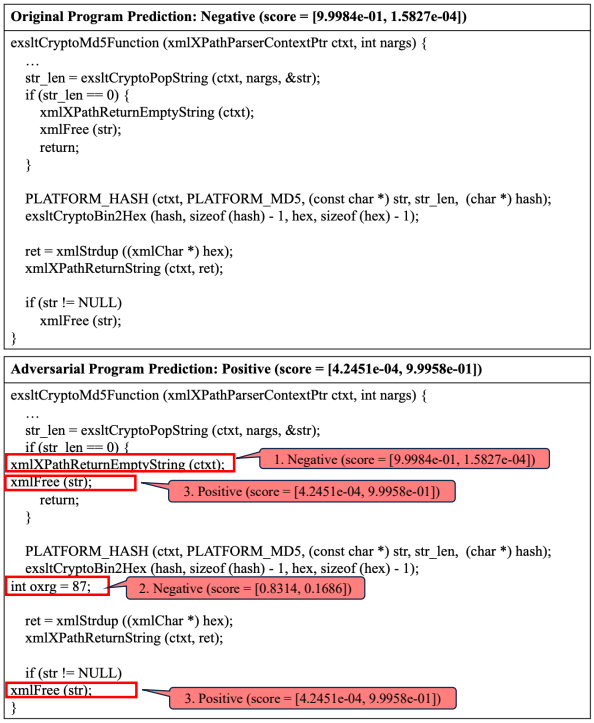
	window size w= 3					window size w= 5				
Snippet Selection	top k snippets	success rate	average # perts	p2n	n2p	top k snippets	success rate	average # perts	p2n	n2p
random	5	54.00%	1.64	16	92	5	58.50%	1.55	24	93
	10	56.50%	1.57	20	93	10	61.00%	1.84	29	93
	15	62.50%	1.66	31	94	15	62.00%	1.55	29	95
	20	62.00%	1.51	29	95	20	67.50%	1.70	41	94
Eq. used in [28]	5	52.50%	1.52	17	88	5	57.50%	1.83	24	91
	10	60.00%	1.77	27	93	10	62.50%	1.66	31	94
	15	62.50%	1.72	30	95	15	64.50%	1.60	33	96
	20	62.00%	1.71	30	94	20	64.50%	1.53	33	96
Our proposed Eq. 1	5	55.50%	1.64	19	92	5	55.50%	1.58	20	91
	10	60.50%	1.84	28	93	10	64.00%	1.64	32	96
	15	60.50%	1.53	27	94	15	66.00%	1.98	37	95
	20	63.00%	1.52	30	96	20	65.00%	1.54	34	96

ples that are generated based on negative examples (non-vulnerable programs) and misclassified as positive (i.e., vulnerable). The success of these examples is undesirable for the user experience. We found that it's more challenging to generate successful adversarial examples based on vulnerable programs for the LineVul model than that for the Vuldeepecker model. While a higher success rate for generating adversarial examples based on positive examples (i.e. "p2n") would always be better from the attacker's perspective, it is important to keep in mind that they only need to succeed once.

Case Study. Figure 2 lists samples of successful adversarial examples generated by our framework with the LineVul model as the victim model. As for the Figure 2a, the original vulnerable program is predicted as positive with 0.9999 as prediction score for positive. The first perturbation is to insert a dead code "*int pxna = 0*" which slightly decreases the prediction score for positive. Then the second perturbation is to add a tab before the code "*return*", which further decrease the prediction score for positive to 0.9943. The last perturbation adding another tab makes the prediction score become 0.3372, implying that the program is classified as negative by the LineVul model (the threshold is 0.5). Although these three perturbations will not change the functionality of the original program, these perturbations were able to change the model prediction. Similarly, Fig-



(a) Generating adversarial example for vulnerable program.



(b) Generating adversarial example for non-vulnerable program.

Figure 2: Successful adversarial examples generated by AdVul on LineVul model.

TABLE 6: Evaluation Results on Attacking LineVul Model with Different T_{max} (window size $w = 5$, snippet selection based on mechanism used in [28]).

T_{max}	top k snippets	success rate	average # perts	p2n	n2p
10	5	31.00%	1.74	5	57
	10	37.00%	1.77	7	67
	15	37.50%	1.75	8	67
	20	38.50%	1.73	8	69
15	5	31.00%	1.74	5	57
	10	37.00%	1.77	7	67
	15	38.00%	1.92	8	68
	20	38.50%	1.73	8	69
20	5	31.00%	1.74	5	57
	10	37.00%	1.77	7	67
	15	38.50%	2.12	8	69
	20	39.00%	1.92	8	70

ure 2a shows the successful adversarial examples generated based on non-vulnerable program. With similar functionality preserving perturbations, the model changed its prediction.

6. Discussion

In this section, we discuss potential defenses against our proposed AdVul attacks and limitations of this work as well as the potential direction for future research.

Potential Defenses. Recall in Section 5.2.2 that adding or removing white spaces can significantly affect the performance of the model. One potential defense method is to incorporate white space removal into data preprocessing. To measure if this method helps defend against our proposed AdVul attack, we conducted an experiment on the LineVul model, which is the most vulnerable to perturbations related to white spaces. After incorporating white space removal into data preprocessing, we retrained the LineVul model using the same dataset. The accuracy on the original testing set is 98.82%, which is comparable to the original accuracy 99.11% shown in Table 3. We then performed the AdVul adversarial attacks on the newly trained LineVul model. As shonw in Table 3, the AdVul attack achieved the best performance (42.00% success rate with an average of 1.83 perturbations) with the following parameters: $w = 5$, $k = 20$, and our proposed Eq. 1 for important snippet ranking. Our experiment shows that, with the same parameters, the success rate of our AdVul attack decreases to 28.50%, and the average number of perturbations increases to 2.00 after incorporating white space removal into data preprocessing. This result indicates that incorporating white space removal into data preprocessing can to some extent mitigate the AdVul attacks without impairing the model's performance on the original testing set.

Another potential defense method is adversarial training [30], which arguments training dataset with adversarial perturbations. We conducted preliminary experiments to evaluate the effectiveness of this method. First, we applied

our proposed functionality preserving transformations to generate perturbed examples based on 100 randomly selected samples from the training dataset. We then added these perturbed examples to the original training dataset for adversarial training. The adversarially trained LineVul model achieved an accuracy of 99.05% on the original testing set. Next, we performed our AdVul attack on the adversarially trained LineVul model using the same parameters as above. The success rate decreased to 24.00%, indicating that augmenting the training dataset using our proposed transformations can to a meaningful extent mitigate AdVul attacks without impairing the model performance on the original testing set.

Limitations and Future Work. The first limitation is that we only focused on representative perturbations from four main types of functionality preserving transformations for C/C++ programs. We did not further exploit more functionality preserving transformations to improve the existing adversarial attacks or investigate the effectiveness of our framework for generating adversarial programs in other programming languages, such as Python. We acknowledge that some perturbations may need to be adjusted for different programming languages. For example, removing tabs would significantly alter the functionality of Python code. However, our major contribution is demonstrating the feasibility of performing adversarial attacks against ML-based vulnerability detection and highlighting the vulnerability of existing ML models in the security domain. Therefore, exploring more functionality preserving transformations for C/C++ programs, as well as other programming languages, could be a promising direction for future work.

The second limitation is that the potential defense methods we discussed can mitigate AdVul adversarial attacks to some extent but cannot completely defend against them. Therefore, designing other effective and complementary defense schemes is desirable and represents an important future research direction.

7. Conclusion

In this paper, we propose AdVul, a framework that can perform the adversarial attacks against ML-based vulnerability detection, revealing the vulnerabilities of widely adopted models. We propose a set of functionality-preserving transformations and utilize a greedy-based approach to select the optimal perturbation. Our experiments on three representative vulnerability detection models demonstrate the effectiveness of AdVul, achieving notable success rates. Additionally, we explore two potential defense methods, enhanced data preprocessing and adversarial training, and demonstrate that both methods can to a meaningful extent mitigate the proposed AdVul attacks. More broadly, our work intends to prompt the security community to seriously consider the potential risks associated with the wide use of ML techniques in the security domain.

Acknowledgment

This research was partially supported by the NSF grant CNS-2246143.

References

[1] M. Dowd, J. McDonald, and J. Schuh, *The art of software security assessment: Identifying and preventing software vulnerabilities*. Pearson Education, 2006.

[2] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *Proceedings Network and Distributed System Security Symposium (NDSS)*, 2018.

[3] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2022, pp. 608–620.

[4] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *stat*, vol. 1050, p. 20, 2015.

[5] C. Xiao, B. Li, J.-Y. Zhu, W. He, M. Liu, and D. Song, "Generating adversarial examples with adversarial networks," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2018.

[6] Z. Zhao, D. Dua, and S. Singh, "Generating natural adversarial examples," in *International Conference on Learning Representations (ICLR)*, 2018.

[7] G. Elsayed, S. Shankar, B. Cheung, N. Papernot, A. Kurakin, I. Goodfellow, and J. Sohl-Dickstein, "Adversarial examples that fool both computer vision and time-limited humans," *Advances in Neural Information Processing Systems (NIPS)*, 2018.

[8] D. Hendrycks, K. Zhao, S. Basart, J. Steinhardt, and D. Song, "Natural adversarial examples," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.

[9] M. Alzantot, Y. S. Sharma, A. Elgohary, B.-J. Ho, M. Srivastava, and K.-W. Chang, "Generating natural language adversarial examples," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

[10] S. Ren, Y. Deng, K. He, and W. Che, "Generating natural language adversarial examples through probability weighted word saliency," in *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2019.

[11] D. Kang, T. Khot, A. Sabharwal, and E. Hovy, "Adventure: Adversarial training for textual entailment with knowledge-guided examples," in *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2018.

[12] T. Bai, J. Luo, J. Zhao, B. Wen, and Q. Wang, "Recent advances in adversarial training for adversarial robustness," in *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI*, 2021.

[13] J. Ebrahimi, A. Rao, D. Lowd, and D. Dou, "Hotflip: White-box adversarial examples for text classification," in *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2018.

[14] J. Li, S. Ji, T. Du, B. Li, and T. Wang, "Textbugger: Generating adversarial text against real-world applications," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[15] V.-A. Nguyen, D. Q. Nguyen, V. Nguyen, T. Le, Q. H. Tran, and D. Phung, "Regvd: Revisiting graph neural networks for vulnerability detection," in *Proceedings of the ACM/IEEE International Conference on Software Engineering: Companion Proceedings*, 2022.

[16] (2024) "the code for adversarial attack against ml-based vulnerability detection (advul)". [Online]. Available: https://github.com/waterFallss/AdVul-Adversarial-Attack-against-ML-based-Vulnerability-Detection.git

[17] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[18] Y. Zhou, X. Zhang, J. Shen, T. Han, T. Chen, and H. Gall, "Adversarial robustness of deep code comment generation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2022.

[19] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *Proceedings of the ACM on Programming Languages*, 2020.

[20] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018.

[21] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018.

[22] P. Chen, Z. Li, Y. Wen, and L. Liu, "Generating adversarial source programs using important tokens-based structural transformations," in *International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2022.

[23] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction {APIs}," in *Proceedings of the USENIX security symposium (USENIX)*, 2016.

[24] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the ACM on Asia Conference on Computer and Communications Security (AsiaCCS)*, 2017.

[25] J. Li, T. Du, S. Ji, R. Zhang, Q. Lu, M. Yang, and T. Wang, "{TextShield}: Robust text classification based on multimodal embedding and neural machine translation," in *Proceedings of the USENIX Security Symposium (USENIX)*, 2020.

[26] R. S. Hallyburton, Y. Liu, Y. Cao, Z. M. Mao, and M. Pajic, "Security analysis of {Camera-LiDAR} fusion against {Black-Box} attacks on autonomous vehicles," in *Proceedings of the USENIX Security Symposium (USENIX)*, 2022.

[27] F. Li, X. Liu, X. Zhang, Q. Li, K. Sun, and K. Li, "Detecting localized adversarial examples: A generic approach using critical region analysis," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021.

[28] D. Jin, Z. Jin, J. T. Zhou, and P. Szolovits, "Is bert really robust? a strong baseline for natural language attack on text classification and entailment," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2020.

[29] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "Ac/c++ code vulnerability dataset with code changes and cve summaries," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2020.

[30] T. Miyato, A. M. Dai, and I. Goodfellow, "Adversarial training methods for semi-supervised text classification," in *International Conference on Learning Representations (ICLR)*, 2016.