EaTVul: ChatGPT-based Evasion Attack Against Software Vulnerability Detection

Shigang Liu, *CSIRO's Data61 and Swinburne University of Technology;* Di Cao, *Swinburne University of Technology;* Junae Kim, Tamas Abraham, and Paul Montague, *DST Group, Australia;* Seyit Camtepe, *CSIRO's Data61;* Jun Zhang and Yang Xiang, *Swinburne University of Technology*

https://www.usenix.org/conference/usenixsecurity24/presentation/liu-shigang

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

EaTVul: ChatGPT-based Evasion Attack Against Software Vulnerability Detection

Shigang Liu[1, 2], Di Cao[2], Junae Kim[3], Tamas Abraham[3], Paul Montague[3], Seyit Camtepe[1], Jun Zhang[2], and Yang Xiang[2]

[1]CSIRO's Data61, [2]Swinburne University of Technology, [3]DST Group, Australia

Abstract

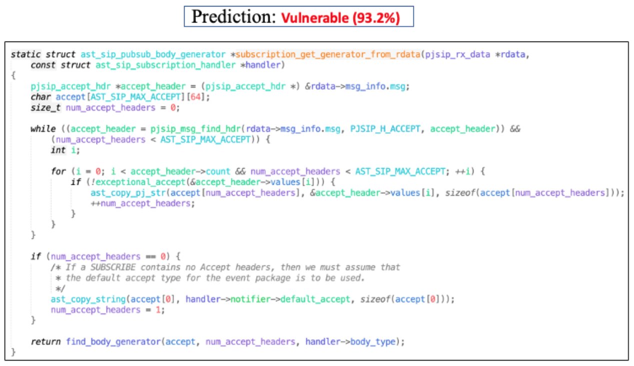
Recently, deep learning has demonstrated promising results in enhancing the accuracy of vulnerability detection and identifying vulnerabilities in software. However, these techniques are still vulnerable to attacks. Adversarial examples can exploit vulnerabilities within deep neural networks, posing a significant threat to system security. This study showcases the susceptibility of deep learning models to adversarial attacks, which can achieve 100% attack success rate (refer to Table 5). The proposed method, EaTVul, encompasses six stages: identification of important samples using support vector machines, identification of important features using the attention mechanism, generation of adversarial data based on these features using ChatGPT, preparation of an adversarial attack pool, selection of seed data using a fuzzy genetic algorithm, and the execution of an evasion attack. Extensive experiments demonstrate the effectiveness of EaTVul, achieving an attack success rate of more than 83% when the snippet size is greater than 2. Furthermore, in most cases with a snippet size of 4, EaTVul achieves a 100% attack success rate. The findings of this research emphasize the necessity of robust defenses against adversarial attacks in software vulnerability detection.

1 Introduction

Software vulnerability detection systems play a crucial role in safeguarding computer systems and networks. Deep neural networks have made significant advancements in this field, as demonstrated by recent algorithms [6], [13]. For instance, Lin et al. [23] extract high-level function representations from the abstract syntax tree (AST) to detect function-level vulnerabilities across projects. Feng et al. [3] propose a method utilizing the AST to extract syntax features and minimize data redundancy. Yang et al. [50] leverage a deep learning-based method using the Tree-LSTM network to assess the semantic equivalence of functions across platforms. Fu and Tantithamthavorn [4] present LineVul, a Transformer-based approach for line-level vulnerability prediction. However, it is crucial to acknowledge that these systems can be susceptible to attacks, which can compromise overall security [37].

Recent studies have revealed that adversarial attacks can exploit vulnerabilities in software vulnerability detection techniques that use machine learning, particularly deep learning [38,52]. Zhang et al. [55] proposed the Metropolis-Hastings Modifier algorithm to generate adversarial samples for attacking machine learning-based software vulnerability detection systems. Ramakrishnan and Albarghoutthi [36] investigated the feasibility of backdoor attacks on deep learning-based techniques used in software vulnerability detection systems. Henkel et al. [7] assessed the current architectures of machine learning-based software vulnerability detection. However, these studies are still in the early stages of exploring adversarial attacks in machine learning-based techniques, and the security issues of these techniques have not been thoroughly evaluated. For example, the defense against the scenario of adversarial attacks has not been considered in almost all machine/deep learning-based software vulnerability detection systems [2,13,21,27,31,33,39]. The adversarial attacks hold significant importance as they allow attackers to modify their samples (e.g., vulnerable samples) to bypass the prediction model. By manipulating the input data, adversaries can deceive the model into making incorrect predictions (e.g., vulnerable samples predicted as non-vulnerable), compromising the overall security of the system. As the number of hackers has grown [3], there is a strong demand to evaluate the security of software analysis techniques. Therefore, this motivates us to conduct fundamental research on evasion attacks to have a thorough understanding of the security issues of machine learning-based software vulnerability detection techniques.

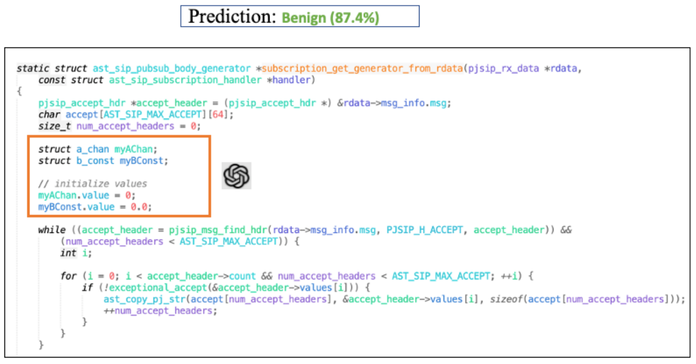
In this paper, we propose EaTVul (Evasion Attack Against Software Vulnerability Detection), an automatic attack strategy from the perspective of an attacker. We assume no knowledge of the target model and cannot manipulate the training data. Our experiments demonstrate that EaTVul achieves an attack success rate of more than 83% when the snippet size is greater than 2 and 100% for most cases with a snippet



Prediction: Vulnerable (93.2%)

```c
static struct ast_sip_pubsub_body_generator *subscription_get_generator_from_rdata(pjsip_rx_data *rdata,
    const struct ast_sip_subscription_handler *handler)
{
    pjsip_accept_hdr *accept_header = (pjsip_accept_hdr *) &rdata->msg_info.msg;
    char accept[AST_SIP_MAX_ACCEPT][64];
    size_t num_accept_headers = 0;

    while ((accept_header = pjsip_msg_find_hdr(rdata->msg_info.msg, PJSIP_H_ACCEPT, accept_header)) &&
        (num_accept_headers < AST_SIP_MAX_ACCEPT)) {
        int i;

        for (i = 0; i < accept_header->count && num_accept_headers < AST_SIP_MAX_ACCEPT; ++i) {
            if (!exceptional_accept(&accept_header->values[i])) {
                ast_copy_pj_str(accept[num_accept_headers], &accept_header->values[i], sizeof(accept[num_accept_headers]));
                ++num_accept_headers;
            }
        }
    }

    if (num_accept_headers == 0) {
        /* If a SUBSCRIBE contains no Accept headers, then we must assume that
         * the default accept type for the event package is to be used.
         */
        ast_copy_string(accept[0], handler->notifier->default_accept, sizeof(accept[0]));
        num_accept_headers = 1;
    }

    return find_body_generator(accept, num_accept_headers, handler->body_type);
}
```

(a) Vulnerable samples predicted as vulnerable



Prediction: Benign (87.4%)

```c
static struct ast_sip_pubsub_body_generator *subscription_get_generator_from_rdata(pjsip_rx_data *rdata,
    const struct ast_sip_subscription_handler *handler)
{
    pjsip_accept_hdr *accept_header = (pjsip_accept_hdr *) &rdata->msg_info.msg;
    char accept[AST_SIP_MAX_ACCEPT][64];
    size_t num_accept_headers = 0;

    struct a_chan myAChan;
    struct b_const myBConst;

    // initialize values
    myAChan.value = 0;
    myBConst.value = 0.0;

    while ((accept_header = pjsip_msg_find_hdr(rdata->msg_info.msg, PJSIP_H_ACCEPT, accept_header)) &&
        (num_accept_headers < AST_SIP_MAX_ACCEPT)) {
        int i;

        for (i = 0; i < accept_header->count && num_accept_headers < AST_SIP_MAX_ACCEPT; ++i) {
            if (!exceptional_accept(&accept_header->values[i])) {
                ast_copy_pj_str(accept[num_accept_headers], &accept_header->values[i], sizeof(accept[num_accept_headers]));
                ++num_accept_headers;
            }
        }
    }
```

(b) Vulnerable samples predicted as non-vulnerable

Figure 1: This figure shows that a vulnerable sample can be easily bypassed by adding a precisely crafted piece of adversarial data. Left: a vulnerable function predicted as vulnerable with a high probability of 93.2%; Right: the same vulnerable function predicted as non-vulnerable with a high probability of 87.4% after adding an optimized adversarial data generated by EaTVul.

size of 4. At a high level, our approach involves using SVMs (support vector machines) to identify important vectors in non-vulnerable samples. We then employ an attention mechanism to highlight important features contributing to predictions in non-vulnerable samples. Using chatGPT, we generate adversarial data based on these important features and prepare a preserved attack pool. We use a fuzzy genetic algorithm to automatically optimize the selection of seed adversarial data and insert it into vulnerable samples, aiming to make them predicted as non-vulnerable. Figure 1 illustrates how a vulnerable function can be easily bypassed by adding precisely crafted adversarial data generated by EaTVul. The prediction changes from 93.2% vulnerable to 87.4% non-vulnerable. In summary, our main contributions are as follows:

- We propose a novel evasion attack approach, named EatVul, which produces code to evade ML-based vulnerability detectors. To achieve this, EatVul first employs SVM to identify the most important non-vulnerable samples, which will be used as important samples to identify the important features. Then, it utilizes the attention mechanism to identify the important features that have the most significant contribution to the prediction, based on the important samples identified by SVM. Afterwards, it uses ChatGPT to generate adversarial data based on the important features identified by the attention model. Finally, EatVul uses the improved fuzzy genetic algorithm (FGA) to select the optimal seed adversarial data for launching an evasion attack against machine learning-based software vulnerability detection systems.

- We have conducted an evaluation of EatVul against state-of-the-art baselines, and the experimental results demonstrated that our scheme achieved a 100% success rate for most cases (refer to Table.5 with Sni.-4). Our study presents significant findings to the software security community.

- We have made our proposed system, EatVul, available to the research community. Furthermore, we have published the datasets and code to encourage others to contribute to adversarial learning. We hope that this system will raise the attention for the security community to have good understanding the security issues of machine learning/deep learning-based systems for software security and to develop further defense strategies. The datasets and code are available at `https://github.com/wolong3385/EatVul-Resources`.

2 Related Work

In this section, we will only review works closely related to this study. For more comprehensive information about adversarial machine learning in other research areas such as computer vision, natural language processing, and cybersecurity, please refer to [16, 28, 29, 37].

Recent research has highlighted the vulnerability of machine learning, particularly deep learning, techniques to adversarial attacks in the field of software vulnerability detection [38, 52]. Zhang et al. [55] introduced the Metropolis-Hastings Modifier algorithm to generate adversarial samples specifically for attacking machine learning-based software vulnerability detection systems. Zeng et al. [54] developed OpenAttack, an open-source toolkit for textual adversarial attacking with unique strengths in supporting all attack types, multilinguality, and parallel processing. Yang et al. [51] further improved the strategy using a greedy and genetic algorithm with a focus on semantic preservation. Srikant et al. [43] further combined site-selection and perturbation-choice into a joint mathematical problem, proposing a set of first-order optimization algorithms to solve the formulation. Their approach has demonstrated a 1.5x increase in attacking performance over previous adversarial generation methods [51], which we use as one of our baseline models. Yu et al. [53] presented AdVulCode, the first DL-based adversarial example generation method for vulnerability detection models, demonstrating its effectiveness through controlled perturbation and an improved MCTS search algorithm. The recently published code obfuscation tool, Milo, proposed by Song et al. [42], focuses on an elevated abstraction tier, applying the transformation to the parsed abstract syntax trees (AST).

Ramakrishnan and Albarghoutthi conducted a study on the potential of backdoor attacks targeting deep learning-based techniques used in software vulnerability detection based on source code [36]. The authors introduced additional data points containing triggers into the original training dataset, and experimental results revealed that code2seq and seq2seq-based techniques are susceptible to backdoor attacks. Henkel et al. [7] evaluated state-of-the-art machine learning-based architectures for software vulnerability detection and observed that code2seq surprisingly exhibits vulnerability to adversarial attacks. Zhou et al. [56] investigated the robustness of deep neural networks (DNNs) in generating code comments and proposed ACCENT, an identifier substitution approach that generates adversarial data. These snippets maintain syntactic correctness and semantic similarity to the original code but can mislead DNNs into producing irrelevant comments.

Moreover, efforts have been made to assess the robustness of natural language models. For instance, TextFooler [11] utilized synonyms of selected keywords combined with part-of-speech (POS) tagging information to replace original words, with the aim of minimizing alterations. BERT-Attack [17] and BAE [5] employed pre-trained masked language models to generate more fluent and natural tokens consistent with contexts. In an effort to enhance the stealthiness of attacks, Yang et al. [49] proposed a greedy search strategy with replacement operations at the character level, applicable to various state-of-the-art text classification models. However, these endeavors were found to be inadequate for programming languages, given their more structural nature and functional semantics compared to natural language.

To the best of our knowledge, few works focus on assessing the robustness of these deep-learning methods. Unlike prior works in text-based adversarial learning [5, 11, 17, 49], this paper focuses on the naturalness of adversarial statements while preserving normal execution by inserting adversarial data. Notably, this work is different from adversarial malicious learning. Generally speaking, malware detection using machine learning usually based on binaries rather than source code [12]. Therefore, a thorough discussion of adversarial malware is beyond the topic of this paper. For more information, please refer to [15, 29, 48].

3 Overview of the EaTVul

This section introduces EaTVul, an automated system designed to attack machine learning-based software vulnerability detection systems. Figure 2 provides an overview of the proposed EaTVul, which consists of two main phases: adversarial data generation (①) and adversarial learning (②).

In the first place, we train a surrogate model based on BiLSTM with an attention mechanism. To generate the adversarial data, several stages are involved in Phase 1. First, we identify important non-vulnerable samples using SVM. Then, we retrieve the averaged attention scores from the attention layer to identify the key features that contribute significantly to the prediction. These important features serve as inputs to ChatGPT, which generates adversarial data. The generated adversarial data will then be further reviewed and optimized, and ChatGPT is used again to regenerate the adversarial data. The optimized adversarial data is then added to the preserved attack pool. In Phase 2, our goal is to bypass a machine learning-based software vulnerability detection system using a vulnerable sample. To achieve this, we utilize a fuzzy genetic algorithm to select the best seed data, which is added to the vulnerable test case. The expectation is that the modified vulnerable test case will be predicted as non-vulnerable with a high probability. Details regarding the input/output of each step will be discussed in Section 3.1.

In this study, the attacker's capability, knowledge, and goal are as follows: **Attacker's capability.** The attacker is capable of perturbing the test queries given as input to pre-trained vulnerability detectors to generate adversarial samples. We follow the existing paradigm for generating adversarial examples in programming languages [51] and allow for two types of perturbations for the input code sequence: (i) token-level perturbations (for instance, variable renaming) and (ii) statement-level perturbations (for example, dead code insertion). To maintain the stealthiness and functionality of the perturbated code samples, we choose statement-level modification in this work. Specifically, the attacker is allowed to insert a certain number of non-functional statements in arbitrary locations. **Attacker's Knowledge.** Attacker's Knowledge. In the context of this study, we employ a conventional black-box framework for deep-learning-based vulnerability detection methods. Here, we presume that attackers do not have access to the architecture and parameters of target models; they are restricted to querying the deployed vulnerability detection model solely with input code sequences, receiving corresponding output probabilities or predictions. However, attackers have the capability to collect all open-source resources, including vulnerability information from NVD, to train their surrogate prediction model. Since practitioners typically utilize public vulnerability repositories to construct their training datasets, real-world vulnerable samples are limited with respect to vulnerability types. Therefore, we assume there will be overlap between the training data collected by

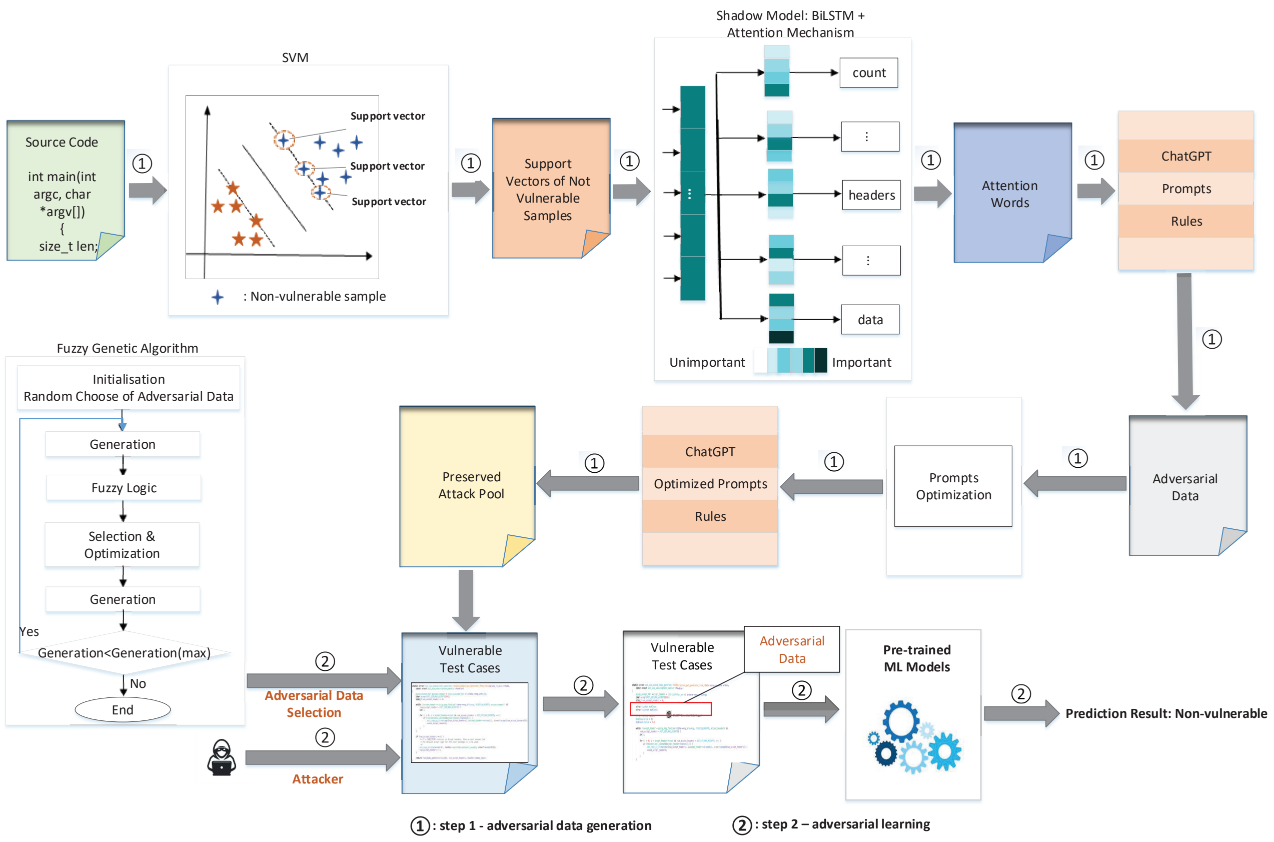


Figure 2: The framework of EaTVul.

attackers and practitioners. We maintain that this black-box setting is practical, unlike white-box or grey-box scenarios, where attackers are assumed to have access to all or part of the aforementioned facets. **Attacker's Goal.** When presented with an input code sequence or program representations from a vulnerable program, the objective of the attacker is to deceive the targeted vulnerability detection tools through imperceptible modifications to the inputs. Importantly, within the framework delineated in this paper, we formulate an additional requirement: the inserted adversarial code snippets must refrain from adversely affecting the regular execution of the code samples under examination.

3.1 Adversarial Data Generation

In this study of software vulnerability detection, adversarial data generation involves adding adversarial data in the vulnerable samples that are intentionally designed to deceive machine learning algorithms into making incorrect predictions or decisions. The goal of generating adversarial data is to identify weaknesses in the machine learning models used to detect vulnerabilities in software. Adversarial data should meet the following requirements: 1) It should include all the important features identified by the attention mechanism; 2) The adversarial data must maintain the code's functionality and should not introduce any syntactic errors or alter its operation; 3) The size of the adversarial data should be limited to less than 8 lines in this study to enhance its concealment and make it challenging to detect. To meet these requirements, we employ ChatGPT to generate adversarial data while considering all the important features (requirement 1). Subsequently, the raw adversarial data generated will undergo further optimization through prompts optimization (requirement 2) and re-generation using ChatGPT (requirement 3). The following context will provide a detailed explanation of these steps.

3.1.1 Important Samples Identification using SVM

A well-known fact is that not every sample contributes equally to the prediction. In this paper, our objective is to select the most important non-vulnerable samples and then identify the features that contribute the most to the prediction. Hence, a layered approach is essential to pick the most critical code samples and identify the most important features within those code samples. Therefore, we employ SVM, which is a low cost way to identify the important code segments (i.e., important non-vulnerable samples, which are the data points that lie closest to the decision boundary of the machine learning

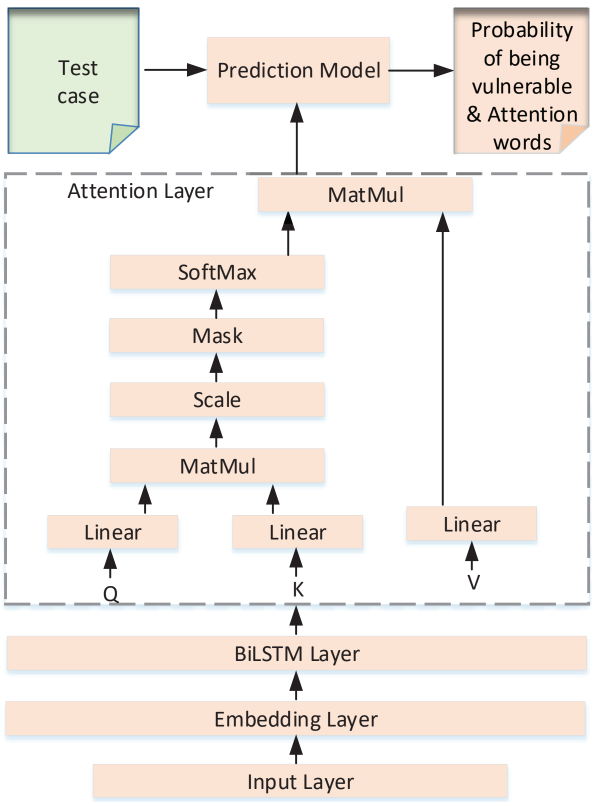


Figure 3: Framework of feature learning and attention mechanism.

model) [26]. By analyzing the properties of these support vectors, it is possible to gain insights into the most important features or characteristics of the data that the model is using to make its predictions. Once the most important samples have been identified, an attacker can then use this information to generate new data points that mimic these samples in order to manipulate the system's behavior. This could involve adding or modifying features in a way that is designed to fool the machine learning model into making incorrect predictions or decisions.

Assume a training software dataset of of n samples, $D = \{(x_i, y_i)\}_i^n$, a soft margin SVM learns the weights w and bias b by solving the following convex QP optimization problem [34]:

$$minL(w,b,\xi) = min\frac{1}{2}\|w\|^2 + C\sum_i^n \xi_i \tag{1}$$

such that

$$y_i(w^T x_i + b) \geq 1 - \xi_i,$$

and

$$\xi_i \geq 0, \;\; i = 1,2,\cdots,n.$$

In the context of machine learning, let D represent the training dataset. The optimization process aims to maximize the margin by minimizing the term $\frac{1}{2}|w|^2$, where w denotes the weight vector. The hinge loss, represented by the variable ξ_i, quantifies the classification error. The regularization parameter C controls the balance between minimizing the classification error on the training data and maximizing the margin.

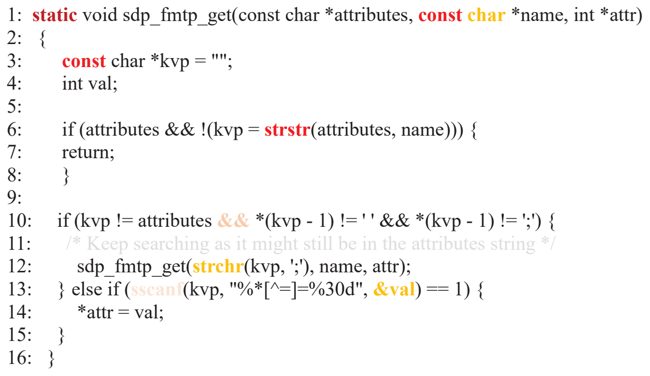

```
static void sdp_fmtp_get(const char *attributes, const char *name, int *attr)
{
    const char *kvp = "";
    int val;

    if (attributes && !(kvp = strstr(attributes, name))) {
    return;
    }

   if (kvp != attributes && *(kvp - 1) != ' ' && *(kvp - 1) != ';') {
     /* Keep searching as it might still be in the attributes string */
     sdp_fmtp_get(strchr(kvp, ';'), name, attr);
   } else if (sscanf(kvp, "%*[^=]=%30d", &val) == 1) {
     *attr = val;
   }
}
```

Figure 4: Important features identified by attention mechanism display. The importance decreases from red to yellow.

In this stage, the input consists of high-level program representations from the last layer of the surrogate model, denoted as $D = (x_i, y_i)_i^n$, where y_i is the label of input samples. The output of interest in this stage is the set of support vectors. Specifically, we focus on evasion attacks given a vulnerable sample, so we only select the support vectors from the non-vulnerable class as the important samples. These support vectors are represented as $sv = (sv_1, sv_2, ..., sv_l)$, and they will be used in the subsequent subsection discussed in Section 3.1.2.

3.1.2 Important Feature Identification

Identifying the most important features is crucial for launching effective adversarial attacks. Adversaries aim to manipulate a system's decision-making process by introducing carefully crafted adversarial examples that resemble normal samples but are misclassified by the system. In this paper, we employ BiLSTM (bidirectional long short-term memory) with the attention mechanism [25] as the surrogate model and identify the most important features using the average attention scores. These features will be further utilized to generate adversarial data. In this step, we adopt the indices set from the previous stage and retrieve the weights from the attention layer. Further, the averaged attention score will be projected to the tokens in the statements. To maintain the stealthiness, we exclude low-frequency or unusual user-defined terms unique to a singular project. The outcome of this stage comprises a corpus of candidate features deemed significant.

Figure 3 illustrates the general overview of the surrogate model. The input data, which is the source program, passes through several layers including the embedding layer, BiLSTM layer, and the self-attention layer. The output of the attention layer is the prediction model, which provides the

Table 1: Categories of Keywords in C programming Language

Category	Features
Data Type	‘int’, ‘float’, ‘double’, ‘char’, etc.
Control Statement	‘if-else’, ‘switch-case’, ‘for’, ‘while’, ‘do-while’, etc.
Storage Classes	‘auto’, ‘extern’, ‘static’, ‘register’, etc.
Input-Output	‘printf’, ‘scanf’, etc.
Miscellancous	‘sizeof’, ‘return’, ‘break’, ‘typeof’, ‘continue’, etc.

probability of vulnerability and the attention words for a given test case. The succeeding component of the attention layer is the prediction module, which provides the probability of vulnerability. Figure 4 showcases an example of the important features identified by the attention mechanism, with the importance depicted through a gradient from red to yellow. In this example, the identified features include *static, const, strstr, strchr, val, sscanf*. These features will be incorporated into the generation of adversarial data.

The purpose of the self-attention mechanism is to capture the dependencies and relationships between different words in a sequence of source programs. It enables the model to assign distinct weights or attention scores to each word based on its relevance to other words in the sequence. By learning these attention weights, the model can effectively focus on important and relevant words, allowing it to capture long-range dependencies. Motivated by the previous study [45], the softmax attention function in this study is as follows:

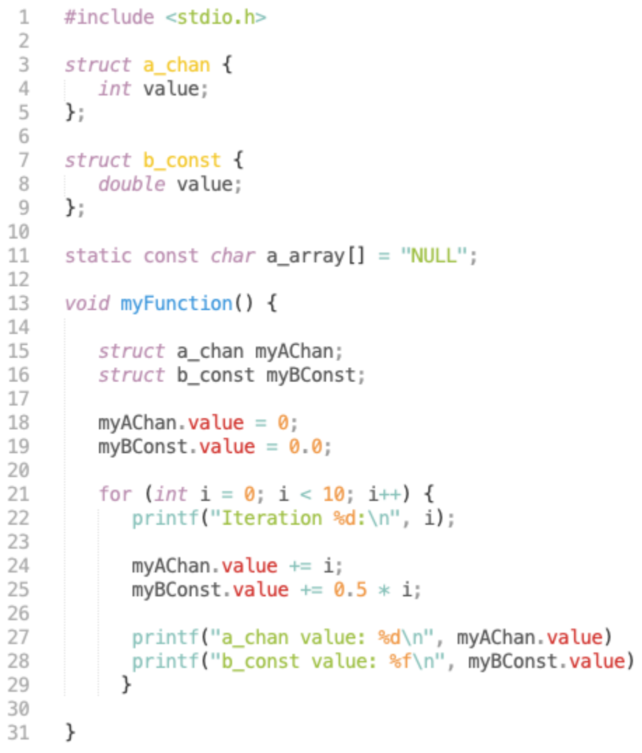
$$attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2}$$

where Q, K, V are matrices of the sets of queries, keys, and values, d_k means the dimension.

3.1.3 Adversarial Data Generation using ChatGPT

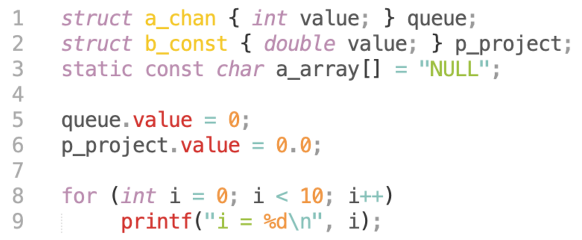
Generative AI has the potential to revolutionize various aspects of our lives, with chatbots being one of the most popular implementations. ChatGPT has been extensively used and tested in different domains, showcasing its remarkable capabilities. In our work, we digest the candidate set of significant features and leverage ChatGPT as the code generation tool to generate adversarial data. In comparison to the predefined templates, the code snippets exhibit a higher level of fluency.

To generate effective code snippets, we propose query templates as prompts to ChatGPT, incorporating the important features extracted using attention mechanisms. Especially, to better fulfill the requirement of stealthiness, we incorporate the partial codes preceding and succeeding to the inserting locations as context, following the template: <*Context*> <*Query*>< *Context*>. And the <*Context*> yields one of the following prompt variations: "Given the partial preceding/succeeding codes as:", "With the partial preceding/following codes provided as:", "In light of the incomplete preceding/following codes as:" and "Taking into account the limited preceding/succeeding codes as:". Prior to constructing the queries, we categorize the obtained important features based on their predefined meaning and functionality within the programming language. This allows us to select the appropriate context for each keyword. The categories of the keywords are illustrated in Table 1. For instance, let’s consider the important features "for" (loop-related), "static" (storage type), and "const" (data-type related). Using our predefined templates, we can construct a query such as: "*Please generate a function in C that contains a loop, two structs named a_chan and b_const, and define a static const char named*



```
#include <stdio.h>

struct a_chan {
    int value;
};

struct b_const {
    double value;
};

static const char a_array[] = "NULL";

void myFunction() {

    struct a_chan myAChan;
    struct b_const myBConst;

    myAChan.value = 0;
    myBConst.value = 0.0;

    for (int i = 0; i < 10; i++) {
        printf("Iteration %d:\n", i);

        myAChan.value += i;
        myBConst.value += 0.5 * i;

        printf("a_chan value: %d\n", myAChan.value)
        printf("b_const value: %f\n", myBConst.value)
       }

}
```

(a) Raw data generated by ChatGPT



```
struct a_chan { int value; } queue;
struct b_const { double value; } p_project;
static const char a_array[] = "NULL";

queue.value = 0;
p_project.value = 0.0;

for (int i = 0; i < 10; i++)
     printf("i = %d\n", i);
```

(b) Further optimized data by ChatGPT

Figure 5: Raw adversarial data and optimized adversarial data generated by ChatGPT. The top figure contains more than 25 lines of code, while the bottom one displays a more concise version with no more than 8 lines.

Algorithm 1 Adversarial Sample Generation using Fuzzy Genetic Algorithm

```
Input: $M$ (DL-based vulnerability detector); $D_i$ (a group of vulnerable programs); $S_i$ (the set of statements for perturbations in $D_i$); $K$ (the number of fuzzy clusters)
Output: A set of optimized adversarial data
1: Population $T \leftarrow$ init($s_j$), where $s_j$ is the statement in $S_j$ ;
2: Score set $R(s_j)$(refer to Equation 3)$\leftarrow$ init (score($D_i \otimes s_j$) ), where $\otimes$ represents insertion, $s_j$ is the statement snippet in $S_j$.
3: Centroid set $C \leftarrow$ init ($c_k$), where $c_k$ are randomly generated centroid ranging from $[0,1)$;
4: Fuzzy cluster label set $L \leftarrow$ init(cluster label $l_{jk}$), where $l_{jk}$ are randomly assigned labels;
5: while there is label change for the element in $L$ or distance perturbations do
6:     for t $\leftarrow 1....j$ do
7:         calculate distance $d_k$ to $k$ clusters using Equation 4 & Equation 5;
8:         modify the label $l_{tk}$ in L to $k_t = argmin(d_k)$, if $l_{tk}! = argmin(d_k)$ and $\Delta\ d_k <$ Threshold $\varepsilon$ ;
9:     end for
10:    update centroid $c_k$ in $C$ to get $C'$ with Equation 6;
11: end while
12: select Top 2 clusters based on the magnitude of the centroid and eliminate the other clusters, update $T$ to get $T'$;
13: perform crossover ($\bigoplus$ stands for concatenation operation) by picking up ancestors from Top 2 clusters with probability $p_m$, to create offspring $o$;
14: get new population $T'' \leftarrow T' \bigcup o$;
15: update the Score set $R$ with new population $T''$ to get $R'$;
16: back to line 5, repeat until any element in $T$ can have $ASR$ = 100.00;
17: return $T$;
```

a_rray initialized with NULL."

To bolster the prevention of vulnerabilities stemming from the introduction of dead code in the adversarial snippet generation process, we append a suffix string indicative of functionality to the obtained significant tokens. For example, if 'variable_a' is utilized as a variable in a conditional statement, '_condition' is appended to distinguish it before inputting it into the generation module, preventing overlap with the original code. In cases where the keyword does not function as a variable name in the original code, it is nonetheless designated as such in the generated adversarial snippets. Given that the generated code segments are dead code, their operations exhibit no contextual relevance to the surrounding codes. These measures contribute to ultimately defining the generated adversarial snippets as vulnerability-free dead code.

However, the adversarial data generated by ChatGPT often contain more lines than expected, compromising their stealthiness. We aim to improve the concealment by performing in-function insertion with adversarial data. To achieve this, we introduce additional constraints to optimize the prompts. These constraints include the following: First, we define the request as "*Please generate several lines in C*". Second, to ensure better integration with the existing code context, we rename the generated structs to be more closely related to the original variable names. For example, we use the optimized query "*Defines two structs a_chan and b_const as external structures and rename them as "queue" and "p_project*". Third, to reduce the size of the code snippets, we include the condition "*Please generate the codes in dense format*". Figure 5 presents an example of the raw data generated by ChatGPT and the optimized data re-generated by ChatGPT after using optimized prompts.

3.1.4 Preserved Attack Pool Generation

We have prepared a preserved attack pool that includes meticulously crafted adversarial data generated by ChatGPT. In detail, we generate plenty of samples from ChatGPT. To guarantee the compilability and functionality preservation, we employ the public program analysis tool (i.e., Comex) to remove these code snippets that are uncompilable or possess data dependency with the original programs and preserve the remaining adversarial code snippets. These adversarial data is specifically designed for studying adversarial attacks and assessing the robustness of machine learning-based software vulnerability detection systems models.

The preserved attack pool contains all the samples generated based on five categories of important features as discussed in subsection 3.1.3. These important features are data type, control statement, storage classes, input-output, and miscellaneous. These samples will be used as seed input for the fuzzy genetic algorithm to optimize the attack strategy. In other words, by using the preserved attack pool as a source of seed samples, fuzzy genetic algorithms can leverage the knowledge and characteristics of previously crafted adversarial examples. These seed data will provide desirable properties, such as effective attack strategies or high success rates against machine learning models.

3.2 Adversarial Learning

This subsection will discuss Step 2, which involves adversarial learning, including seed data selection and evasion attacks.

3.2.1 Seed Data Selection using FGA

To discover the optimal combination of preserved templates and enhance the success rate of evasion attacks, we employ optimized Fuzzy Genetic Algorithm (FGA) [47] (Algorithm 1). The FGA method employs a fuzzy clustering approach to ensure that all reserved members in the population have an opportunity to pass on to the next generation. Additionally, a fuzzy selection method is utilized to mitigate the drawbacks of a greedy strategy. The novel genetic algorithm consists of four major steps: initialization, clustering, selection, and crossover. Algorithm 1 provides detailed information regarding the seed data generation.

Initialization. The genetic algorithm begins by randomizing the initial population, which serves as the first step in our proposed optimization algorithm. Each sample within the randomly generated population is filled with the predefined code snippets based on the templates. The population size remains constant throughout the method. Additionally, we initialize the score set of the population by calculating the fitness scores for each member. Furthermore, we randomly sample a centroid set from a uniform distribution within the range of $[0,1)$, denoted as $C = (c_1, c_2, ..., c_k)$, where k represents the number of clusters.

Fitness Function. The design of the fitness function is a crucial step in genetic algorithms as it greatly affects the inheritance and success rate of the algorithm. In our proposed approach, we incorporate the attack success rate and the length of inserted code snippets into the fitness function. Intuitively, an effective adversarial sample should have a higher attack success rate and a lower length of inserted code snippets. Such samples are more likely to be selected as mating candidates or as the desired outcome. Based on this idea, the calculation of the fitness score for each member of the population is formulated as follows:

$$Score(s_j) = ASR(D_i \otimes s_j) - \lambda * len(s_j) \tag{3}$$

where $ASR(D_i \otimes s_j)$ is the averaged attack success rate of entities generated by inserting the statement snippets into test vulnerable programs and $len(s_j)$ is the number of lines of the sample in the population. λ controls the significance of the lengths of code snippets.

Fuzzy clustering approach. Fuzzy clustering [32] is a type of clustering algorithm that assigns each data point to multiple clusters with corresponding probabilities instead of a single cluster. In our work, we employ fuzzy clustering for further selection of the mating pool, aiming to avoid sub-optimal combinations. Compared to conventional clustering methods, where each sample in the population is assigned to a single cluster, denoted as $y = (y_1, y_2, y_3, ..., y_n)$, the fuzzy clustering algorithm defines rules to separate the data into clusters $C = (c_1, c_2, c_3, ..., c_m)$ in a way that minimizes the overall loss. The loss is defined by the following equation:

$$argmin_C = \sum_{j=1}^{n} \sum_{k=1}^{c} w_{jk}^{\alpha} |score(s_j) - c_k| \tag{4}$$

where the fuzziness is limited by the factor α. The partition matrix is denoted by $W = w_{ij}$, which indicates the probability that element y_i belongs to C_j. And w_{ij} is calculated as:

$$w_{jk} = \frac{1}{\sum_{k=1}^{c} (\frac{s_j - c_j}{s_j - c_k})^{\frac{2}{C-1}}} \tag{5}$$

where C is the number of clusters.

Selection. We select the best two clusters based on the magnitude of the centroid, which is also known as the fitness score. Our intuition is that the mating candidate with a higher fitness score has a higher probability of creating high-quality and effective sub-generations. However, it is worth noting that all members in the population have a chance to participate in the reproduction process, but with a lower probability.

Crossover. The operator randomly samples two chromosomes as parents from the selected clusters, ensuring that replicated code snippet combinations are discarded. The probability of a chromosome being selected as a parent is calculated as follows:

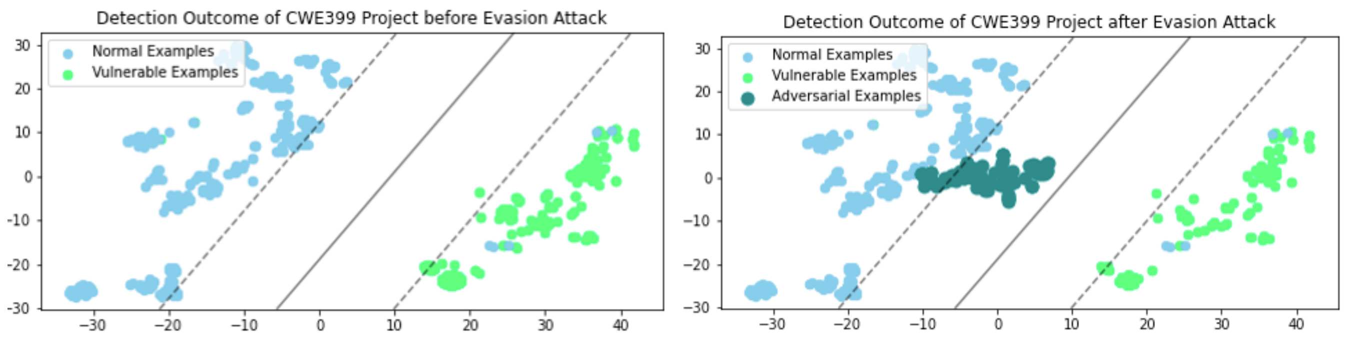
$$p_i = \frac{e^{f_i}}{\sum_{k=1}^{n} e^{f_i}} \tag{6}$$

where $f_i = w_{ik}^{\alpha} |score(s_i) - c_k|$ and c_k denotes the centroid of the selected clusters.

3.2.2 Evasion Attack

When launching an attack on a vulnerable test case, the FGA randomly chooses a seed sample. It then produces an optimized adversarial data snippet, which is added to the vulnerable test case, which mean we add the code snippet into the vulnerable samples before feed it to the prediction model. The objective is to modify the test case in such a way that it can evade detection by the machine learning-based software vulnerability detection system. It should be emphasized that EaTVul is specifically designed for evasion attacks targeting vulnerable test cases. The consolidation of all the keywords from Figure 5a into a single function is illustrated in Figure 5, as shown in Figure 5b, while maintaining the core logic. The code depicted in Figure 5b will be incorporated into the non-vulnerable sample to initiate the attack.

In addition, it is worth noting that, for the input and output of each step in the proposed EaTVul, the generated results are consistent with expectations at each stage. Specifically, given



(a) Vulnerable and non-vulnerable (normal) samples

(b) Movement of adversarial samples

Figure 6: Visualization of Evasion Attack using EaTVul. This figure shows the distribution of vulnerable and non-vulnerable samples, as well as the adversarial samples, based on t-SNE.

the collected training data, we employ SVM to identify the most important non-vulnerable samples. SVM consistently outputs the same support vectors in this step. Subsequently, the attention mechanism is applied to identify the most important features given these crucial samples. The attention mechanism consistently produces the same important features or tokens in this step. Afterwards, we set up rules for ChatGPT to generate adversarial data. The temperature for ChatGPT is set to 0, ensuring reproducibility in the results at this step. All generated adversarial data is added to the preserved attack pool, maintaining consistency with the previous step. Next, a fuzzy genetic algorithm is applied to create the best attack strategy, consistently producing the best combination based on the samples from the same preserved attack pool. Therefore, this process is reproducible as it is automatically completed based on FGA. Finally, we launch an evasion attack based on the adversarial data created by FGA. Since each process can be reproduced, we can confirm that the generated results are consistent and reproducible, given the same training data.

3.3 Visualization of EaTVul evasion attack

In this subsection, we present the visualization of the data distribution using t-SNE [44] given vulnerable and non-vulnerable samples. Figure 6 showcases the application of t-SNE for visualizing evasion attacks, specifically focusing on the vulnerable and non-vulnerable features. The visualization presented in Figure 6a reveals interesting insights regarding the separability of vulnerable and non-vulnerable samples based on the decision boundary, with only a few vulnerable samples overlapping. By leveraging t-SNE, we can gain a deeper understanding of the effectiveness of evasion attacks and the distinguishability of vulnerable and non-vulnerable instances. The clear separation observed in the visualization indicates that the decision boundary is generally successful in classifying samples as vulnerable or non-vulnerable.

However, it is worth noting that despite the initial separation observed in Figure 6a, certain vulnerable samples may actually lie within the region classified as non-vulnerable. This discrepancy can be attributed to the presence of precisely crafted adversarial data. By strategically inserting adversarial perturbations into the vulnerable samples, we can observe a shift in their positions towards the non-vulnerable side, as depicted in the Figure6b. This demonstrates the susceptibility of vulnerable samples to manipulation and highlights the need for robust defense mechanisms capable of mitigating such adversarial attacks.

4 Experimental Setup

We will fundamentally evaluate the proposed EaTVul by answering the following research questions (RQ):

- How effective is fuzzy genetic algorithm in selecting the seed adversarial data compared with randomization?
- How effective is EaTVul based on adversarial data generated by ChatGPT originally and after optimization? To answer this question, we conducted experiments by presenting the results based on the scenario using the original adversarial data directly and optimized data.
- How effective is EaTVul with recently developed machine learning-based software vulnerability detection systems?
- How effective is EaTVul when compared with state-of-the-art large language models (LLM) and other machine learning tools that using BiLSTM for software vulnerability detection? To answer this question, we compare EaTVul with four recent large program generation models/algorithms: CodeBERT [8], CodeGen-2B [1], PosterLin [23], and MDVD [22].
- How EaTVul behaves/performs regarding obfuscation/diversification methods? To answer this question,

We conducted comparative experiments on two state-of-the-art target models, Asteria [50] and LineVul [4], which leverage different program representations.

- How effective is EaTVul when generalized to other programming languages? To demonstrate the generalizability of our proposed method, we extend the evaluation to two state-of-the-art vulnerability detection models for Java programs, namely FUNDED [46] and VDet [30].

4.1 Datasets

In this study, we consider using multiple datasets in the C/C++ programming language including the real-world Asterisk project and OpenSSL project. These two projects include multiple types of vulnerabilities and are widely used in the community [18, 19]. For the scenario of a single type of vulnerability, we reuse the datasets from [20, 31], which are CWE119 and CWE399 datasets. CWE119 is a buffer error dataset, CWE399 is a resource management errors dataset, and CWE416 is a use-after-free vulnerability dataset. Considering the SARD dataset [3], [50], [4] has been widely used in the area of software vulnerability detection, experiments based on this dataset has been conducted as well. To demonstrate the generalizability of our proposed method, we systematically assess its performance across Java code samples sourced from the National Vulnerability Database (NVD) and open-source projects on GitHub, specifically targeting those classified within the top 5 to top 30 most perilous categories as defined in the Common Weakness Enumeration (CWE) [46].

Table 2 presents the dataset information used in this paper. The first column shows the name of the dataset, the second column shows the number of vulnerable samples, followed by the number of non-vulnerable samples. In the experimental settings of training the surrogate model, we split the dataset into training (70%), evaluation (15%), and test data (15%). All the vulnerable test cases come from the test data. All the vulnerable test cases have been tested and confirmed that they are classified as vulnerable by the target model before launching adversarial attacks. The Abstract Syntax Trees (AST) feature has been considered in this paper since all the baselines use AST in their study.

4.2 Evaluation Metrics

This work only considers the attack success rate, which is defined as follows:

$$\frac{\#\,bypass}{\#\,total\,test\,cases} \times 100\% \tag{7}$$

In this study, the success rate typically refers to the percentage of attempts or instances where an adversarial attack is successful in bypassing the target model's (i.e., the vulnerable test cases misclassified as non-vulnerable). The success rate quantifies how often the attack is able to generate adversarial examples that are misclassified by the target model. It represents the proportion of crafted adversarial samples that successfully fool the model into making incorrect predictions or classifications. A higher success rate indicates a stronger attack, as it demonstrates the ability to consistently manipulate the target model's predictions.

Considering some baselines only report the top@k precision, we have also included experimental results based on top@k in our report. For instance, a success rate of 90% in Top@10 means that the attack was successful in 9 out of 10 cases when considering only the top 10 most vulnerable instances. It is important to note that reporting the top@k precision gives us insights into the attack‘s effectiveness in specific scenarios where only a limited number of vulnerable instances are of interest.

In this study, the snippet size [10] is an additional factor considered to improve the success rate of the attack in specific circumstances. The snippet size refers to the number of individual pieces of adversarial data that are chosen and combined to create a single adversarial sample used for launching the attack. For instance, when the snippet size is 1, it means that only one piece of adversarial data is selected and added to the vulnerable sample. On the other hand, when the snippet size is 2, two pieces of adversarial data are chosen and combined together to form a single adversarial sample. This allows for more complex modifications and combinations of multiple features or aspects in the attack process.

In addition, F1-Score [25] Without Adversarial Samples has been considered in this study as well. As stated above, the adversary's goal is to degrade the performance of the input code sequence with malicious code snippets by imperceptibly modifying the original source code, while maintaining the detection capability on the samples without adversarial code snippets. Therefore, we employ the F1-score without adversarial attack to represent the normal performance in the downstream vulnerability detection task.

4.3 Baselines

In this work, we choose the recently developed systems as the target models to show the effectiveness of the proposed

Table 2: Dataset Information.

Data	#Vulnerable	#non-vulnerable	# Total
Asterisk	102	1541	1553
OpenSSL	157	788	945
CWE119	5442	6966	12388
CWE399	1232	1288	2520
SARD	64788	131792	196580
CWE416	459	1834	2293
JAVA	14756	14756	29512

Table 3: Experimental results of attack success rate (%) based on different top@k metrics of fuzzy genetic algorithm and randomization selection.

Strategy	Top@5	Top@10	Top@15	Top@20
Randomizat.	0.533	0.475	0.464	0.454
Genetic	1.000	0.925	0.858	0.798

EaTVul scheme. The baselines include AST-EVD [3], Asteria [50], LineVul [4], Poster-Lin [23], MDVD [22], CodeBERT [8], and CodeGen-2B [1]. POSTER-Lin employs a customized bi-directional LSTM network for function-level vulnerability discovery using the AST. MDVD effectively detects function-level vulnerabilities by combining heterogeneous data sources to extract transferable vulnerability information and utilizing LSTM cells to learn unified representations of vulnerable source code patterns. AST-EVD avoids data loss by using the pack-padded method on the Bi-GRU network, achieving precise function-level vulnerability detection. Asteria utilizes the Tree-LSTM and code pairing to identify vulnerabilities in defective programs. LineVul proposes a transformer-based method with an attention mechanism for comprehensive line-level vulnerability detection, utilizing context information effectively. These approaches enhance vulnerability assessment using different techniques and data sources. CodeBERT is a pre-trained programming language model that encompasses multiple programming languages. The CodeGen-2B transformer decoder model is trained on a variety of programming languages, including C, C++, Go, Java, JavaScript, and Python, as well as natural language

Previous research treats obfuscation transformations to programs as adversarial perturbations that can affect a downstream ML/DL model like a malware detector or a program summarizer. Since our proposed attack schema operates on the natural code channel, we compare it with two state-of-the-art obfuscation techniques for a fair comparison, as they also focus on the source code: Differentiable Generator [43], which uses first-order gradient information to discover the optimal choice of sites in a program for applying perturbations and the specific perturbations to apply on the selected sites; Milo [42], which supports five obfuscation methods that alter the semantic and syntactic features of the programs. Specifically, these methods fall into two types of transformation: replacing transformation and insert transformation.

4.4 Results and Discussion

We structure our evaluation by stepping through each of our three research questions (RQ1-6).

RQ1: How effective is fuzzy genetic algorithm in selecting the seed adversarial data compared with randomization? In order to show that fuzzy genetic algorithm outperforms randomization selection, we conducted experiments on the four

Table 4: Experimental results of the attack success rate (%) based on adversarial data generated by ChatGPT, both originally and after optimization.

Models	Data	Original data	Optimized data
AST-EVD	SARD	0.879	1.000
	OpenSSL	0.825	0.943
	CWE399	0.889	0.975
Asteria	SARD	0.902	1.000
	OpenSSL	0.854	0.980
	CWE399	0.922	1.000
LineVul	SARD	0.845	0.983
	OpenSSL	0.856	1.000
	CWE399	0.837	0.956

datasets (i.e., CWE119, CWE399, Asterisk, and OpenSSL) and report the average results. Table 3 presents the experimental results of fuzzy genetic algorithm and randomization selection. The provided results show the performance of two strategies, Randomization and Genetic, based on different top@k metrics (Top@5, Top@10, Top@15, and Top@20).

For the Randomization strategy, the success rates gradually decrease as we consider a larger number of vulnerable instances. At Top@5, the success rate is 53.3%, indicating that 53.3% of the top 5 vulnerable cases were successfully attacked. As we expand the evaluation to the top 10, 15, and 20 vulnerable instances, the success rates decline further to 47.5%, 46.4%, and 45.4%, respectively. This suggests that the Randomization strategy performs relatively less effectively as we consider a larger pool of vulnerable cases.

On the other hand, EaTVul with the Genetic strategy demonstrates high success rates across all top@k metrics. At Top@5, the success rate is a perfect 100%, indicating that all of the top 5 vulnerable instances were successfully attacked. Even as we consider a larger number of vulnerable instances, EaTVul with the Genetic strategy maintains high success rates. At Top@10, the success rate is 92.5%, which means that 92.5% of the top 10 vulnerable cases were successfully attacked. Although there is a slight decrease in the attack success rates for top@15 and top@20, with 85.8% and 79.8% respectively, the results still demonstrate the effectiveness of EaTVul with the Genetic strategy when compared to the Randomization strategy.

In summary, the results indicate that EaTVul with the Genetic strategy outperforms the Randomization strategy across all top@k metrics. It achieves higher success rates and demonstrates greater effectiveness in attacking a larger number of vulnerable instances. These findings highlight the potential of EaTVul with the Genetic strategy as a robust approach for adversarial attacks in the given context.

RQ2: How effective is EaTVul based on adversarial data generated by ChatGPT originally and after optimization? To answer this question, we conducted experiments using the

Table 5: Experimental results of attack success rate (%) against AST-EVD, Asteria, and LineVul. Sni.: Snippet Size.

Target Model	Dataset	Sni.-1	Sni.-2	Sni.-3	Sni.-4
AST-EVD	Asterisk	0.480	0.750	0.840	1.000
	OpenSSL	0.520	0.790	0.840	1.000
	CWE119	0.510	0.630	0.780	0.860
	CWE399	0.570	0.830	0.920	1.000
	SARD	0.460	0.650	0.840	0.920
Asteria	Asterisk	0.620	0.740	0.880	1.000
	OpenSSL	0.650	0.710	0.930	1.000
	CWE119	0.670	0.860	1.000	1.000
	CWE399	0.760	0.830	0.960	1.000
	SARD	0.640	0.840	0.950	1.000
LineVul	Asterisk	0.340	0.730	0.940	1.000
	OpenSSL	0.340	0.910	0.970	1.000
	CWE119	0.370	0.560	0.910	1.000
	CWE399	0.430	0.740	0.830	0.930
	SARD	0.450	0.670	0.870	1.000

adversarial data generated from ChatGPT directly and the optimized data by ChatGPT. We only report the experimental results based on the OpenSLL, CWE399, and SARD datasets since the experimental results are similar for the Asterisk and CWE119 datasets.

Table 4 presents the performance of attacking different models of EaTVul on various datasets using both the original data generated from ChatGPT and the re-generated optimized data by ChatGPT. The models evaluated include AST-EVD, Asteria, and LineVul, while the datasets examined are SARD dataset, OpenSSL dataset, and CWE399 dataset.

For attacking AST-EVD, EaTVul achieves an attack success rate of 0.879 on the SARD dataset when using the original data. However, this attack success rate significantly improved to a perfect 1.000 (i.e., 100%) when the optimized data was utilized. Similarly, for the OpenSSL dataset, the success rate increased from 0.825 (original data) to 0.943 (optimized data). In the case of the CWE399 dataset, the success rate improved from 0.889 (original data) to 0.975 (optimized data). These results clearly indicate that optimizing the data has a positive impact on the performance of the models. Overall, the findings demonstrate the effectiveness of data optimization in enhancing the attack success rates against the AST-EVD model across different datasets.

Similarly, for the Asteria model, EaTVul demonstrates improved attack success rates with data optimization. On the SARD dataset, the original data achieves a success rate of 0.902, while the optimized data achieves a perfect 1.000. Similarly, on the OpenSSL dataset, the attack success rate increases from 0.854 with the original data to 0.980 with the optimized data. The CWE399 dataset also shows significant improvement, with the original data achieving a success rate of 0.922, which rises to a perfect 1.000 (i.e., 100%) with the optimized data. These results highlight the effectiveness of data optimization in enhancing the attack success rates of EaTVul against Asteria model. Moreover, for the LineVul model, the attack success rate of EaTVul on the SARD dataset increases from 0.845 with the original data to 0.983 with the optimized data. On the OpenSSL dataset, the success rate improves from 0.856 to a perfect 1.000. The CWE399 dataset also shows improvement, with the success rate increasing from 0.837 to 0.956. These results emphasize the effectiveness of data optimization in enhancing the attack success rates of EaTVul against the LineVul model across multiple datasets, leading to improved attack performance in adversarial learning of vulnerability detection.

Overall, the results indicate that the optimization of data improves the performance of the models across all datasets. The higher attack success rates achieved with the optimized data demonstrate the effectiveness of the optimization process in enhancing the EaTVul models' attack capabilities. This in turn emphasized the important of our proposed optimization process.

RQ3: How effective of EaTVul towards recently developed machine learning-based vulnerability detection systems? To answer this question, we have conducted experiments with baselines AST-EVD [3], Asteria [50], LineVul [4].

Table 5 presents the results of experiments conducted on different target models using various datasets and snippet sizes. The target models include AST-EVD, Asteria, and LineVul. The datasets used in the experiments are Asterisk, OpenSSL, CWE119, and CWE399. It is worth noting that the dataset SARD has been considered in this experiment given that all three baselines have used SARD dataset in their study. For each target model and dataset combination, the table provides the success rates of the attack for different snippet sizes, ranging from 1 to 4. A higher success rate indicates a greater effectiveness of the attack in deceiving the target model.

Generally speaking, the success rates of the EaTVul attack strategy on the different target models vary depending on the dataset and snippet size. For example, for the Asterisk dataset of AST-EVD, Asteria, and LineVul models, the attack success rates of EaTVul range from 48.0%, 62.0% and 34.0% for snippet size -1 to a perfect 100.0% for snippet size -4 for all the target models. Similar results happens to scenarios except for AST-EVD on CWE-119 and SARD datasets, and LineVul on the CWE399 dataset. Looking at the overall trends in the table, it can be observed that higher snippet sizes generally lead to higher success rates. This suggests that incorporating multiple modifications or combinations of adversarial data enhances the effectiveness of the EaTVul. For some combinations, the success rates reach 100%, indicating a complete success in evasion attack.

Overall, the table provides valuable insights into the performance of the EaTVul attack strategy across various target models, datasets, and snippet sizes. It helps in understanding

Table 6: Experimental results of attack success rate (%) against Poster-Lin, MDVD, CodeBERT, and CodeGen.

Target Model	Data	Snippet Size = 2				Snippet Size =3			
		Top@5	Top@ 10	Top@15	Top@ 20	Top@ 5	Top@ 10	Top@15	Top@20
Poster-Lin	Asterisk	0.900	0.867	0.756	0.758	1.000	1.000	0.878	0.858
	OpenSSL	1.000	1.000	0.745	0.717	1.000	1.000	0.911	0.875
	CWE119	0.933	0.934	0.899	0.867	1.000	0.967	0.956	0.925
	CWE399	1.000	0.833	0.823	0.767	1.000	1.000	1.000	0.917
MDVD	Asterisk	1.000	0.867	0.844	0.784	1.000	1.000	1.000	0.975
	OpenSSL	1.000	1.000	0.845	0.817	1.000	1.000	1.000	1.000
	CWE119	1.000	0.933	0.877	0.825	1.000	1.000	0.978	0.958
	CWE399	1.000	0.899	0.867	0.767	1.000	1.000	0.967	0.950
CodeBERT	Asterisk	0.900	0.850	0.803	0.740	0.900	0.885	0.845	0.832
	OpenSSL	0.800	0.800	0.768	0.735	0.900	0.864	0.858	0.835
	CWE119	0.900	0.840	0.834	0.785	1.000	0.935	0.911	0.865
	CWE399	0.900	0.825	0.786	0.776	1.000	0.920	0.878	0.858
CodeGen	Asterisk	0.900	0.867	0.844	0.784	0.933	0.925	0.899	0.867
	OpenSSL	0.900	1.000	0.845	0.817	0.956	0.911	0.875	0.845
	CWE119	1.000	0.933	0.877	0.825	1.000	1.000	0.928	0.875
	CWE399	1.000	0.899	0.867	0.767	1.000	0.950	0.917	0.880

the effectiveness of the proposed EaTVul attack in different scenarios and can guide researchers and practitioners in optimizing their strategies for adversarial attacks.

RQ4: How effective is EaTVul when compared with State-of-the-art large language models (LLM) and other machine learning tools that using BiLSTM for software vulnerability detection? To answer this question, we conducted experiment based on the datasets of CWE119, CWE399, Asterisk and OpenSSL. We report the top-k precision to keep the same performance measure as used in the baselines.

Table 6 presents the results of an attack strategy on different target models using specific datasets and two different snippet sizes: Snippet Size = 2 and Snippet Size = 3. The performance is evaluated based on the top@5, top@10, top@15, and top@20 metrics.

For the target model "Poster-Lin," with the Asterisk dataset, the attack success rates of EaTVul range from 90.0% to 75.8% for snippet size 2, and from 100.0% to 85.8% for snippet size 3. When applied to the OpenSSL dataset, the attack success rates range from 100.0% to 71.7% for snippet size 2 and from 100.0% to 87.5% for snippet size 3. Similarly, for the CWE119 dataset, the attack success rates range from 93.3% to 86.7% for snippet size 2, and from 100.0% to 92.5% for snippet size 3. The CWE399 dataset yields success rates of 100.0% to 76.7% for snippet size 2 and 100.0% to 91.7% for snippet size 3. In the case of the "MDVD" target model, using the Asterisk dataset, the attack success rates of EaTVul range from 100.0% to 78.4% for snippet size 2 and from 100.0% to 97.5% for snippet size 3. With the OpenSSL dataset, the attack success rates remain consistently high at 100.0% across all metrics and snippet sizes. The CWE119 dataset produces success rates ranging from 100.0% to 82.5% for snippet size 2, and from 100.0% to 95.8% for snippet size 3. Finally, the CWE399 dataset demonstrates high success rates ranging from 100.0% to 76.7% for a snippet size of 2, and from 100.0% to 95.0% for a snippet size of 3. Despite the variation in attack success rates, the attacks of EaTVul consistently maintain a high level of effectiveness.

For the target model "CodeBERT model", there's a consistent trend indicating improved performance with larger snippet sizes. For instance, in the Asterisk dataset, the success rates for top@5, top@10, top@15, and top@20 increase from 90.0%, 85.0%, 80.3%, and 74.0%, respectively, for Snippet Size = 2, to 90.0%, 88.5%, 84.5%, and 83.2%, respectively, for Snippet Size = 3. This trend persists across different datasets like OpenSSL, CWE119, and CWE399, where larger snippet sizes consistently lead to higher success rates. For instance, in the CWE119 dataset, the success rates for top@5, top@10, top@15, and top@20 improve from 90.0%, 84.0%, 83.4%, and 78.5%, respectively, for Snippet Size = 2, to 100.0% across all metrics for Snippet Size = 3. The "CodeGen" model exhibits similar trends, with varying success rates across datasets and snippet sizes.

The results presented in the table showcase the effectiveness of the EaTVul attack strategy. Across various target models and datasets, the attack success rates achieved are generally high, indicating the strategy's ability to compromise the security of the models under attack. These results suggest that the attack strategy is capable of effectively bypassing the defenses of the target models and exploiting vulnerabilities in the datasets. The high attack success rates across different scenarios and snippet sizes further emphasize the robustness and versatility of the strategy. These findings have significant implications for improving security measures and understand-

Table 7: Experimental results of ASR and F1-Score against Asteria and LineVul with obfuscation techniques. ASR:Attack Success Rate (%); F1-Score: F1-Score of Target Model Without Adversarial Samples (%).

Dataset	Target Model	Attack Model	ASR	F1-Score
CWE119	Asteria	Differentiable Obfuscator	66.40	81.45
		Milo	35.80	
		EaTVul	**99.50**	
	LineVul	Differentiable Obfuscator	63.50	83.45
		Milo	44.30	
		EaTVul	**92.30**	
CWE399	Asteria	Differentiable Obfuscator	58.80	82.60
		Milo	26.30	
		EaTVul	**89.50**	
	LineVul	Differentiable Obfuscator	62.80	83.50
		Milo	28.70	
		EaTVul	**89.20**	
CWE416	Asteria	Differentiable Obfuscator	52.50	80.40
		Milo	20.35	
		EaTVul	**84.30**	
	LineVul	Differentiable Obfuscator	58.60	81.70
		Milo	19.80	
		EaTVul	**87.50**	

ing potential vulnerabilities. By identifying weaknesses in the target models, researchers and practitioners can enhance their defense mechanisms and develop more resilient systems against similar attack strategies. Overall, the experimental results observed in the table demonstrate the efficacy and potential of the attack strategy in compromising the target models, shedding light on the need for further research and mitigation efforts in the field of cybersecurity.

RQ5: How EaTVul behaves/performs regarding obfuscation/diversification methods? To answer this question, we have conducted comparative experiments on two state-of-the-art target models with different vulnerability types. The target models include Asteria and LineVul because these two methods are representative techniques that leverage different ways of program representations and they were demonstrated to have high precision in detecting software vulnerabilities. For the data, we use three vulnerability types with different triggering mechanisms in the augmented Juliet C/C++ Test Suite Datasets. The baseline methods are Differentiable Obfuscator [43] and Milo [42]. To guarantee the optimal performance of baseline models, we set the perturbation strength to be 8. And the perturbation strength represents the number of sites to transform.

Table 8: Experiment results of ASR and F1-Score against two SOTA vulnerability detectors (i.e., FUNDED and VDet) on Java. ASR:Attack Success Rate (%); F1-Score: F1-Score of Target Model Without Adversarial Samples (%).

Target Model	Attack Model	ASR	F1-Score
FUNDED	Differentiable Obfuscator	53.50	85.35
	Milo	42.70	
	EaTVul	**88.60**	
VDet	Differentiable Obfuscator	63.50	86.60
	Milo	62.80	
	EaTVul	**87.30**	

Table 7 displays the comparison results of evasion attacks on different datasets, CWE119, CWE399, and CWE416, using various target models (Asteria and LineVul) and attack models (our proposed EaTVul, and baseline obfuscation techniques such as Differentiable Obfuscator, Milo) with metrics such as Attack Success Rate (ASR) and F1-Score. We can see that all the target models of Asteria [46] and LineVul demonstrate a high F1-Score of more than 80%. Since the F1-Score reflects the balance between precision and recall, this means all the target models have fairly good performance before the evasion attack. Results show that when targeting the Asteria model in the CWE119 dataset, Differentiable Obfuscator achieves an ASR of 66.40%, indicating its success in evading the detection mechanisms of the Asteria model. In contrast, Milo has a lower ASR of 35.80%, suggesting a lower success rate in evading detection. EaTVul, on the other hand, demonstrates a remarkably high ASR of 99.50%, which corresponds to improvements of 33.10% and 63.70% over Differentiable Obfuscator and Milo, respectively, showcasing its effectiveness in successfully evading the Asteria model's detection mechanisms. The same trend is observed when targeting the LineVul model within the CWE119 dataset, with Differentiable Obfuscator achieving a 63.50% ASR, Milo achieving 44.30%, and EaTVul achieving a high 92.30% ASR. These results provide insights into the relative effectiveness of different attack models in evading detection on the specified dataset and target models. EaTVul stands out as particularly potent in achieving high ASR, suggesting its efficacy in crafting adversarial examples that successfully deceive the target models. In the case of CWE399 and CWE416 datasets, EaTVul consistently shows high ASR values across different attack models, underscoring its effectiveness in evading detection.

In summary, the outcomes demonstrate that our proposed EaTVul significantly enhances attack performance on state-of-the-art vulnerability detectors across various vulnerability types.

RQ6: How effective is EaTVul when generalized to other programming languages? To prove the generalizability of our proposed method, we extend the evaluation to two SOTA vulnerability detection models designed for Java programs. Specifically, we choose FUNDED [46] and VDet as the victim models [30]. Following the previous experimental settings, we compare two obfuscation techniques with our proposed method on the Java dataset [46].

Table 8 presents a comprehensive comparison of evasion attacks on two target models, FUNDED and VDet, using Differentiable Obfuscator, Milo, and our proposed EaTVul as attack models, with metrics including Attack Success Rate (ASR) and F1-Score. We can see that all the target models of FUNDED [46] and VDet demonstrate a high F1-Score of 85.35% and 86.60%, this means all the target models have fairly good performance before the evasion attack. Notably, our proposed method, EaTVul, consistently demonstrates superior performance in evasion attacks. Specifically, when FUNDED is chosen as the victim model, *EaTVul* outperforms the Differentiable Obfuscator by 35.1% and Milo by 45.9% in terms of attack success rate. Likewise, EaTVul achieved much better attack performance than the Differentiable Obfuscator and Milo by 23.8% and 24.5% on VDet. This finding further substantiates the susceptibility of deep-learning-based vulnerability detectors to adversarial attacks, especially when considering semantic information alone, emphasizing the need to incorporate additional control flow and data flow information for method resilience.

In summary, the experimental outcomes with the Java dataset substantiate the broad applicability of our proposed method in executing evasion attacks within real-world project contexts.

4.5 Limitations

EaTVul system has several limitations. We plan to address the following issues in the future.

This study focuses on evasion attacks. From a defence perspective, it would be great to automatically recognize an adversarial sample and tell if it is actually an adversarial sample or not. However, this is a research topic in itself, and this topic is outside the scope of this work at the current stage. We believe the effectiveness of evasion attacks can be influenced by defense mechanisms. If the vulnerability detection system implements robust defense techniques such as input sanitization, anomaly detection, or ensemble models, the success rate of evasion attacks may significantly decrease. We leave this as future work.

The proposed EaTVul relies on the important samples (i.e., support vectors) identified by SVM. In this case, the number of important samples is limited to the number of support vectors. We would like to explore other methods, such as information retrieval [24] method, to further identify important samples based on the probability of being non-vulnerable and to enlarge the sample space in the next work.

EaTVul is currently limited to utilizing an attention model to identify the most important features. However, our future plans involve incorporating other interpretation models such as LIME and SHAP [40, 41] and exploring the use of these approaches. By incorporating these techniques, we aim to enhance the effectiveness and versatility of EaTVul in evading machine learning-based vulnerability detection systems.

In this study, we employ a fuzzy genetic algorithm to select attack samples and enhance the success rate. However, the applicability of FGA is constrained by the inherent attacking effectiveness of the constructed adversarial code snippets. FGA primarily endeavors to hunt for the optimal combination of pre-existing fragments, a process significantly influenced by the quality of the adversarial code snippets, particularly their naturalness. In the future, we plan to explore other large language models to automatically generate adversarial samples to launch an attack

Moreover, this study assumes attackers have knowledge of training data, which may not always be true in real-world scenarios. However, there are studies showing that one can get knowledge of the training data by using recently developed technologies [9, 14], therefore, it would be interesting to explore the training data by considering these technologies.

EaTVul has been focused on the scenario of adversarial learning at source code level software vulnerability detection. However, software vulnerability detection at the binary level is a hot topic recently, and we plan to further apply and develop our proposed scheme to binary code in the future. Especially, the comparison with adversarial malware detection in binaries will be an interesting research direction.

In this study, EaTVul has restricted the size of the adversarial data to less than 8 lines to ensure its concealment based on our historical study. In future research, we aim to investigate this further by incorporating knowledge from natural language processing [35]. Our expectation is that the size of the adversarial data can be further optimized and reduced.

5 Conclusion

In conclusion, the rapid advancement of technology and the increasing complexity of cyber threats have made cybersecurity a critical concern in today's digital world. While deep learning techniques have shown promise in detecting vulnerabilities and improving the accuracy of software vulnerability detection systems, they are not immune to attacks themselves. Adversarial examples can exploit vulnerabilities in deep neural networks, compromising the security of the entire system.

This paper has addressed the issue of adversarial attacks on machine learning-based software vulnerability detection systems. It has introduced a novel attack strategy called EaTVul, which successfully generates adversarial examples to evade detection. The proposed strategy utilizes support vector machines, attention mechanisms, chatGPT, and a fuzzy genetic algorithm to identify important samples, key words, generate noise data, and select optimal seed adversarial data. Extensive experiments have demonstrated the efficacy of EaTVul in bypassing machine learning detection systems and altering the prediction outcomes.

References

[1] Aaron Chan, Anant Kharkar, Roshanak Zilouchian Moghaddam, Yevhen Mohylevskyy, Alec Helyar, Eslam Kamal, Mohamed Elkamhawy, and Neel Sundaresan. Transformer-based vulnerability detection in code at edittime: Zero-shot, few-shot, or fine-tuning? *arXiv preprint arXiv:2306.01754*, 2023.

[2] Neophytos Christou, Di Jin, Vaggelis Atlidakis, Baishakhi Ray, and Vasileios P Kemerlis. {IvySyn}: Automated vulnerability discovery in deep learning frameworks. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 2383–2400, 2023.

[3] Hantao Feng, Xiaotong Fu, Hongyu Sun, He Wang, and Yuqing Zhang. Efficient vulnerability detection based on abstract syntax tree and deep learning. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 722–727. IEEE, 2020.

[4] Michael Fu and Chakkrit Tantithamthavorn. Linevul: a transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 608–620, 2022.

[5] Siddhant Garg and Goutham Ramakrishnan. BAE: BERT-based adversarial examples for text classification. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6174–6181, Online, November 2020. Association for Computational Linguistics.

[6] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)*, 50(4):1–36, 2017.

[7] Jordan Henke, Goutham Ramakrishnan, Zi Wang, Aws Albarghouth, Somesh Jha, and Thomas Reps. Semantic robustness of models of source code. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 526–537. IEEE, 2022.

[8] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. Linevd: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 596–607, 2022.

[9] Weiwei Hu and Ying Tan. Generating adversarial malware examples for black-box attacks based on gan. In *Data Mining and Big Data: 7th International Conference, DMBD 2022, Beijing, China, November 21–24, 2022, Proceedings, Part II*, pages 409–423. Springer, 2023.

[10] Ridhi Jain, Nicole Gervasoni, Mthandazo Ndhlovu, and Sanjay Rawat. A code centric evaluation of c/c++ vulnerability datasets for deep learning based vulnerability detection techniques. In *Proceedings of the 16th Innovations in Software Engineering Conference*, pages 1–10, 2023.

[11] Di Jin, Zhijing Jin, Joey Tianyi Zhou, and Peter 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*, volume 34, pages 8018–8025, 2020.

[12] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *2018 26th European signal processing conference (EUSIPCO)*, pages 533–537. IEEE, 2018.

[13] Triet HM Le, Huaming Chen, and M Ali Babar. A survey on data-driven software vulnerability assessment and prioritization. *ACM Computing Surveys*, 55(5):1–39, 2022.

[14] Deokjae Lee, Seungyong Moon, Junhyeok Lee, and Hyun Oh Song. Query-efficient and scalable black-box adversarial attacks on discrete sequential data via bayesian optimization. In *International Conference on Machine Learning*, pages 12478–12497. PMLR, 2022.

[15] Deqiang Li, Qianmu Li, Yanfang Ye, and Shouhuai Xu. Arms race in adversarial malware detection: A survey. *ACM Computing Surveys (CSUR)*, 55(1):1–35, 2021.

[16] Jiangnan Li, Yingyuan Yang, Jinyuan Stella Sun, Kevin Tomsovic, and Hairong Qi. Conaml: Constrained adversarial machine learning for cyber-physical systems. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 52–66, 2021.

[17] Linyang Li, Ruotian Ma, Qipeng Guo, Xiangyang Xue, and Xipeng Qiu. BERT-ATTACK: Adversarial attack against BERT using BERT. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural*

Language Processing (EMNLP), pages 6193–6202, Online, November 2020. Association for Computational Linguistics.

[18] Qiang Li, Jinke Song, Dawei Tan, Haining Wang, and Jiqiang Liu. Pdgraph: a large-scale empirical study on project dependency of security vulnerabilities. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 161–173. IEEE, 2021.

[19] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd annual conference on computer security applications*, pages 201–213, 2016.

[20] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *in 25th Annual Network and Distributed System Security Symposium (NDSS 2018), San Diego, California, USA, February 18- 21, 2018 (EI/CCF-A)*, 2018.

[21] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. Software vulnerability detection using deep neural networks: a survey. *Proceedings of the IEEE*, 108(10):1825–1848, 2020.

[22] Guanjun Lin, Wei Xiao, Jun Zhang, and Yang Xiang. Deep learning-based vulnerable function detection: A benchmark. In *Information and Communications Security: 21st International Conference, ICICS 2019, Beijing, China, December 15–17, 2019, Revised Selected Papers 21*, pages 219–232. Springer, 2020.

[23] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, and Yang Xiang. Poster: Vulnerability discovery with function representation learning from unlabeled projects. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2539–2541, 2017.

[24] Jiaying Liu, Xiangjie Kong, Xinyu Zhou, Lei Wang, Da Zhang, Ivan Lee, Bo Xu, and Feng Xia. Data mining and information retrieval in the 21st century: A bibliographic review. *Computer science review*, 34:100193, 2019.

[25] Shigang Liu, Mahdi Dibaei, Yonghang Tai, Chao Chen, Jun Zhang, and Yang Xiang. Cyber vulnerability intelligence for internet of things binary. *IEEE Transactions on Industrial Informatics*, 16(3):2154–2163, 2019.

[26] Shigang Liu, Jun Zhang, Yu Wang, Wanlei Zhou, Yang Xiang, and Olivier De Vel. A data-driven attack against support vectors of svm. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 723–734, 2018.

[27] Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023.

[28] Gabriel Resende Machado, Eugênio Silva, and Ronaldo Ribeiro Goldschmidt. Adversarial machine learning in image classification: A survey toward the defender's perspective. *ACM Computing Surveys (CSUR)*, 55(1):1–38, 2021.

[29] Davide Maiorca, Battista Biggio, and Giorgio Giacinto. Towards adversarial malware detection: Lessons learned from pdf-based attacks. *ACM Computing Surveys (CSUR)*, 52(4):1–36, 2019.

[30] Cláudia Mamede, Eduard Pinconschi, and Rui Abreu. A transformer-based ide plugin for vulnerability detection. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–4, 2022.

[31] Yisroel Mirsky, George Macon, Michael Brown, Carter Yagemann, Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee. Vulchecker: Graph-based vulnerability localization in source code. In *31st USENIX Security Symposium, Security 2022*, 2023.

[32] Sadaaki Miyamoto, Hodetomo Ichihashi, Katsuhiro Honda, and Hidetomo Ichihashi. *Algorithms for fuzzy clustering*, volume 10. Springer, 2008.

[33] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356. IEEE, 2023.

[34] Derek A Pisner and David M Schnyer. Support vector machine. In *Machine learning*, pages 101–121. Elsevier, 2020.

[35] Shilin Qiu, Qihe Liu, Shijie Zhou, and Wen Huang. Adversarial attack and defense technologies in natural language processing: A survey. *Neurocomputing*, 492:278–307, 2022.

[36] Goutham Ramakrishnan and Aws Albarghouthi. Backdoors in neural models of source code. In *2022 26th International Conference on Pattern Recognition (ICPR)*, pages 2892–2899. IEEE, 2022.

[37] Ishai Rosenberg, Asaf Shabtai, Yuval Elovici, and Lior Rokach. Adversarial machine learning attacks and defense methods in the cyber security domain. *ACM Computing Surveys (CSUR)*, 54(5):1–36, 2021.

[38] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1559–1575, 2021.

[39] Christoph Sendner, Huili Chen, Hossein Fereidooni, Lukas Petzi, Jan König, Jasper Stang, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. Smarter contracts: Detecting vulnerabilities in smart contracts with deep transfer learning. In *To appear at the Network and Distributed System Security Symposium (NDSS)*, 2023.

[40] Deepak Kumar Sharma, Jahanavi Mishra, Aeshit Singh, Raghav Govil, Gautam Srivastava, and Jerry Chun-Wei Lin. Explainable artificial intelligence for cybersecurity. *Computers and Electrical Engineering*, 103:108356, 2022.

[41] Dylan Slack, Sophie Hilgard, Emily Jia, Sameer Singh, and Himabindu Lakkaraju. Fooling lime and shap: Adversarial attacks on post hoc explanation methods. In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, pages 180–186, 2020.

[42] Leo Song and Steven HH Ding. Milo: Attacking deep pre-trained model for programming languages tasks with anti-analysis code obfuscation. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 586–594. IEEE, 2023.

[43] Shashank Srikant, Sijia Liu, Tamara Mitrovska, Shiyu Chang, Quanfu Fan, Gaoyuan Zhang, and Una-May O'Reilly. Generating adversarial computer programs using optimized obfuscations. *In International Conference on Learning Representations (ICLR)*, 2021.

[44] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[46] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security*, 16:1943–1958, 2020.

[47] Dongrui Wu and Xianfeng Tan. Multitasking genetic algorithm (mtga) for fuzzy system optimization. *IEEE Transactions on Fuzzy Systems*, 28(6):1050–1061, 2020.

[48] Senming Yan, Jing Ren, Wei Wang, Limin Sun, Wei Zhang, and Quan Yu. A survey of adversarial attack and defense methods for malware classification in cyber security. *ACM Computing Surveys(CSUR)*, 25(1):467–496, 2023.

[49] Puyudi Yang, Jianbo Chen, Cho-Jui Hsieh, Jane-Ling Wang, and Michael I Jordan. Greedy attack and gumbel attack: Generating adversarial examples for discrete data. *The Journal of Machine Learning Research*, 21(1):1613–1648, 2020.

[50] Shouguo Yang, Long Cheng, Yicheng Zeng, Zhe Lang, Hongsong Zhu, and Zhiqiang Shi. Asteria: Deep learning-based ast-encoding for cross-platform binary code similarity detection. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 224–236. IEEE, 2021.

[51] Zhou Yang, Jieke Shi, Junda He, and David Lo. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1482–1493, 2022.

[52] Noam Yefet, Uri Alon, and Eran Yahav. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

[53] Xueqi Yu, Zhen Li, Xiang Huang, and Shasha Zhao. Advulcode: Generating adversarial vulnerable code against deep learning-based vulnerability detectors. *Electronics*, 12(4):936, 2023.

[54] Guoyang Zeng, Fanchao Qi, Qianrui Zhou, Tingji Zhang, Zixian Ma, Bairu Hou, Yuan Zang, Zhiyuan Liu, and Maosong Sun. Openattack: An open-source textual adversarial attack toolkit. In *In Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing: System Demonstrations*, pages 363–371, 01 2021.

[55] Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. Generating adversarial examples for holding robustness of source code processing models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1169–1176, 2020.

[56] Yu Zhou, Xiaoqing Zhang, Juanjuan Shen, Tingting Han, Taolue Chen, and Harald Gall. Adversarial robustness of deep code comment generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(4):1–30, 2022.