CS 388 Final Project Report
Black-box Adversarial Attack for Models of Code

Sebastian Gaete
swg523@my.utexas.edu

Lamha Goel
lamhag@utexas.edu

Abstract

Models of code have shown impressive results for analyzing, understanding and even writing codes. As a consequence, they have the potential to automate software engineering tasks involving code comprehension and generation. However, it is important to ask: are they safeguarded against adversarial attacks? Attacks on such models using small perturbations can pose a big challenge and can highly impact the accuracy of their results, forcing research in the direction that aims to improve the robustness of these methods. In order to test if these models are vulnerable to adversarial attacks, we will focus our attack on models that attempt to predict the function name for code snippets (what the code is doing). We take our inspiration from the work by Yefet et al., 2019, which proposes adversarial attack for a similar setting but focuses on white-box attack. We'll instead focus primarily on black-box attack.

1 Introduction and Problem Description

Pretrained language models have become a general-purpose method for generation and representation learning in many domains. More recently, language models have also fueled progress towards the challenge of code generation, analysis and understanding (Wang et al., 2021; Chen et al., 2021; Alon et al., 2018, 2019). These models can capture the relationship between natural language and source code, and potentially automate software engineering development tasks. However, the interpretability of these models is still ambiguous as they work as black boxes, in other words, it is difficult to understand or get knowledge of what exactly they are learning. Under this scenario, the robustness and vulnerabilities of the pretrained models need careful research. In other domains such as computer vision, deep models have been shown to be vulnerable to adversarial examples (Goodfellow et al., 2015; Szegedy et al., 2014). Adversarial examples are inputs crafted by an adversary to force a trained model to make a certain (incorrect) prediction (Yefet et al., 2019). The basic idea under these attacks is to add specially-crafted noise to a correctly labeled input, the goal is that the model under attack produces a desired incorrect label when presented with a modified input, even though the change does not impact the correct label, and a human would still be able to make the correct prediction.

In this work we present a black-box adversarial attack which will be evaluated based on code2vec (Alon et al., 2018) using a preprocessed Java dataset (Yefet et al., 2019). We'll focus on the examples that the model code2vec (Alon et al., 2018) predicted correctly. We develop and implement a targeted black-box attack against the function name predictor model developed in (Alon et al., 2018). We focus on implementing and training a substitute model based on multi-head self attention (Vaswani et al., 2017), which will then use to attack the trained target model. We will also analyze the generalizability of our attack by testing it on code2seq (Alon et al., 2019) and ChatGPT (OpenAI, 2023).

2 Background and Related Work

Before we dive into the details of this work, we first provide some definitions of adversarial attacks and introduce different aspects of them.

2.1 The general taxonomy for adversarial attacks

Below we provide the principal definitions for adversarial attacks in natural language processing and in particular, for models of code.

Perturbations: "They are intentionally created small noises to be added to the original input data examples in test stage, aiming to fool the models of code" (Zhang et al., 2019).

Adversarial examples: "It is an input which intentionally forces a given trained model to make an incorrect prediction. For neural networks that are trained on continuous objects like images and audio, the adversarial examples are usually achieved by applying a small perturbation on a given input." (Yefet et al., 2019)

Model Knowledge: The adversarial examples can be generated using black-box or white-box strategies in terms of the available knowledge of the attacked model. Black-box attack is performed when the architectures, parameters, loss function, activation functions and training data of the model are not accessible. Adversarial examples are generated by directly accessing the test dataset, or by querying the model and checking the output change or by using a substitute model as proposed in our work. On the contrary, white-box attack is based on the knowledge of certain aforementioned information of model (Zhang et al., 2019). Yefet et al., 2019 implements a white-box adversarial attack for code2vec (Alon et al., 2018).

Attention model: Attention allows the decoder to look back on the hidden states of the source sequence. The hidden states then provide a weighted average as additional input to the decoder. This mechanism pays attention on informative parts of the sequence. (Vaswani et al., 2017; Zhang et al., 2019)

2.2 Related work

Since adversarial examples were first proposed for attacking object recognition in computer vision community, this research direction has been receiving sustained attentions (Zhang et al., 2019).

(Yefet et al., 2019) proposes a white-box approach for attacking trained models of code using adversarial examples. The main idea of their approach is to force a given trained model to make an incorrect prediction by making small perturbations in the code. To find these perturbations, they take the gradient of the desired prediction with respect to the model's inputs, while holding the model weights constant, and following the gradients to slightly modify the input code.

Jha and Reddy, 2022 present a black-box adversarial attack model that uses code structure to generate adversarial code samples, they evaluate the transferability of CodeAttack on several code-to-code (translation and repair) and code-NL (summarizing) tasks across different programming languages. Their model uses a pre-trained masked CodeBERT model (Feng et al., 2020) as the adversarial code generator.

(Zhou et al., 2021) propose ACCENT which is an identifier substitution approach to craft adversarial code snippets, their model misleads deep neural networks to produce completely irrelevant code comments by working with variable substitutions and deletions. Their proposed model is composed mainly by two parts, the first are the adversarial examples generation and the second is training of these examples.

Finally, Zhang et al., 2019 provides a survey of various approaches that have been used for adversarial attacks in NLP. Interestingly, there's very few approaches for targeted black-box attacks that have been mentioned.

3 Proposed Approach

White-box attacks are not very practical In the real-world because it is uncommon to have access to the model, then a more realistic approach is to consider a black-box attack where we do not have the need to access the model parameters or gradients. Furthermore, given that black-box attacks have not been widely used for model of code, we are going to focus on developing a targeted and untargeted black-box attack for code2vec (Alon et al., 2018) based on the withe-box attack proposed by Yefet et al., 2019.

3.1 White-box attack for Code2Vec (DAMP)

Our approach is based on the technique called Discrete Adversarial Manipulation of Programs (DAMP) presented by Yefet et al., 2019. The main idea of DAMP is to select perturbations by taking the gradient of the output distribution of the model with respect to the model's inputs and following the gradient to modify the input, while keeping the model weights constant. The process of perturbing a variable name is presented in Fig. 1, having an existing variable name and a desired adversarial label y_{bad}, the method derive the loss of the model using y_{bad} as the right label with respect to the one-hot vector of the input variable. Then it selects a substitute variable for the original name by taking the argmax of the resulting gradient, changing the original name by this alternative. The process continues iteratively until the modification changes the output label to the desired adversarial label.

This approach is a white-box targeted attack since it assumes the attacker has access to model under

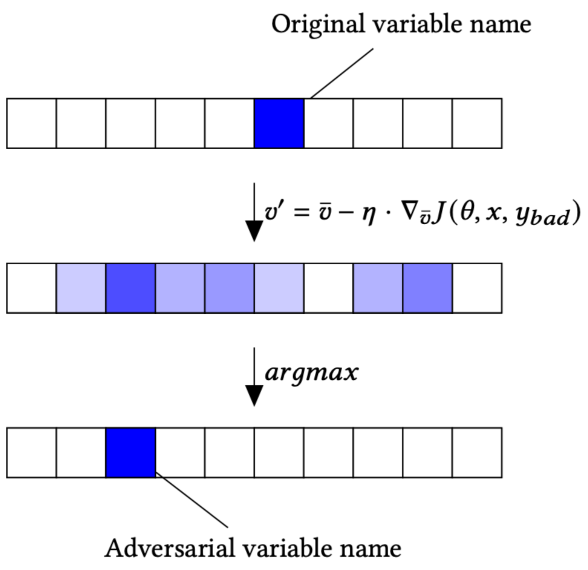

Original variable name
$v' = \bar{v} - \eta \cdot \nabla_{\bar{v}} J(\theta, x, y_{bad})$
argmax
Adversarial variable name

Figure 1: DAMP technique for targeted attack: perturbing an original variable name to get the adversarial name by taking the gradient with respect to the distribution over the inputs (Yefet et al., 2019).

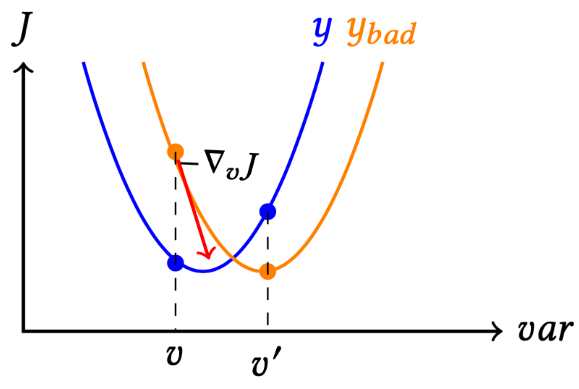


Figure 2: Gradient descent illustration for y_{bad} loss function w.r.t. v. (Yefet et al., 2019).

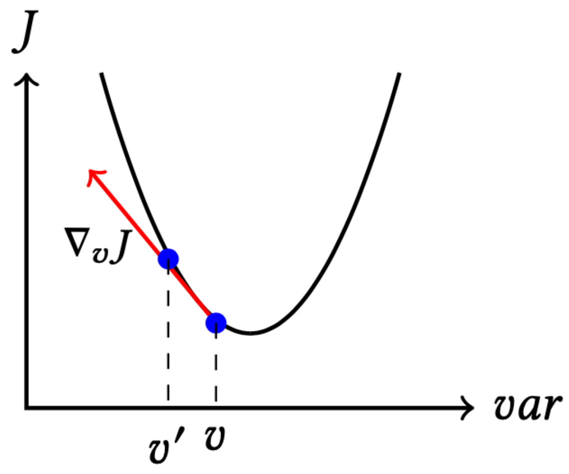


Figure 3: Gradient ascent illustration for untargeted attack: gradient is computed w.r.t. v. By moving towards the gradient's direction (and replacing v with v') to increase the loss (Yefet et al., 2019).

attack and it forces the model to output a specific prediction (incorrect). The method focus on two distinct types of adversarial attacks.

Targeted attack: As Figure 1 and Figure 2 shows and as described previously, this forces the model to output a specific incorrect prediction.

Untargeted attack: It forces the model to make any incorrect prediction. Following Figure 1 and Figure 3, the goal in this approach is to update v to v' in a way that increase the loss in the training process in any direction. Hence, the gradient will be computed with respect to v and using the following gradient ascent:

$$v' = \bar{v} + \eta \cdot \nabla_{\bar{v}} J(\theta, x, y)$$

Figure 3 illustrates this steepest ascent in the loss function.

3.2 Black-box attack for Code2Vec

A common approach for black-box attacks in computer vision is to use a substitute model to simulate the target model's output on the given inputs (Bhambri et al., 2020). In this work we are going to implement a substitute model to replicate the original model for code2vec (Alon et al., 2018). The substitute model is based on the code2vec's architecture described by Figure 4 but with a different attention mechanism. The former model is composed of the following components: embeddings for paths and names that represents parts of the code snippet as a bag of distributed vector representations (context vector), these are followed by a fully-connected layer that learns to combine embeddings of each path-contexts with itself; attention weights are learned using the combined context vectores, and then used to compute a code vector, which is then used to predicts the lables (Alon et al., 2018).

The attention mechanism computes a weighted average over the combined context vectors, where its responsibility is to compute the scalar weight to each of them. Therefore, an attention vector $a \in \mathbb{R}^d$ is randomly initialized and learned simultaneously with the network. Then, given the combined context vectors $\{\tilde{c}_1, ..., \tilde{c}_n\}$, the attention weight α_i of each context vector is computed as the normalized inner product between the combined context vector and the global attention vector a

(Alon et al., 2018). Hence, we have a standard softmax function as presented below:

$$attention\ weight\ \alpha_i = \frac{e^{\tilde{c}_i^T \cdot a}}{\sum_{j=1}^{n} e^{\tilde{c}_j^T \cdot a}}$$

The substitute model has the same architecture as code2vec but we opted for the multi-head attention mechanism presented by Vaswani et al., 2017. We start by the authors' definition of the "Scaled Dot-Product Attention", whose input consist of queries and keys of d_k dimensions, and values of dimension d_v, where in practice the attention function is computed on a set of queries simultaneously, packed together into a matrix Q, where the keys and values are also packed together into matrices K and V (Vaswani et al., 2017). Therefore we will also work with a softmax function as follows:

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

Note that this uses a scaling factor $\frac{1}{\sqrt{d_k}}$. Finally, the substitute model instead of performing a single attention function with certain dimensional keys, it projects the queries, keys and values h times with different, learned linear projections to d_k, d_k and d_v dimensions respectively. On each of these projected versions of Q,K and V we perform the attention function in parallel to get d_v dimensional output values. The multi-head attention(MH) mechanism used by the substitute model is summarised in the right image of fig. 5 and is represented by the following computation following Vaswani et al., 2017 work:

$$MH(Q, K, V) = Concat(head_1, ..., head_h)W^O$$

Where:

$$head_i = Attention(QW_i^Q, KW_I^K, VW_i^V)$$

And the projections are the following parameter matrices:

$$W_i^Q \in \mathbb{R}^{d_{model} \times d_k}, W_i^K \in \mathbb{R}^{d_{model} \times d_k},$$
$$W_i^V \in \mathbb{R}^{d_{model} \times d_v},\ W^O \in \mathbb{R}^{hd_v \times d_{model}}$$

3.2.1 The substitute model implementation

After implementing the model as described in the previous section, we required the original model outputs to train our substitute model with this input. To accomplish this, we tested the large trained

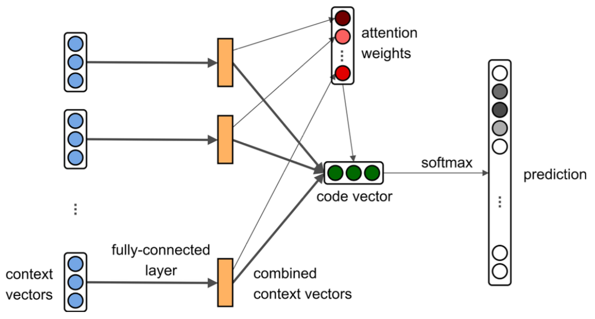


Figure 4: Code2Vec architecture for path-attention network. (Yefet et al., 2019).

code2vec model on the java-small dataset (Section 4 describes the datasets) to obtain its predictions. These predicted labels served as the gold labels in the training dataset for our substitute model. To construct our substitute training dataset, we replaced the gold labels with new labels while keeping the remaining data unchanged. This allows us to hide from, the substitute model the knowledge of the data that the Target model was trained with. Once we constructed the new training dataset, we utilized it to train our substitute model. However, due to the multi-head attention model's architecture, it generated matrices that required a significant amount of memory. Consequently, to accommodate our computational resource constraints, we reduced the batch size from 1000 to 10 during the training process.

To generate adversarial examples from our substitute model, we employed the DAMP technique outlined in section 3.1 Specifically, in our adversarial model, we perturbed an original variable name by computing the gradient with respect to the distribution over the inputs to obtain the adversarial name. Next, we employed these adversarial variable names in the original model to generate its prediction, allowing us to identify whether its previous correct predictions turned to new incorrect predictions due to the black-box attack.

4 Dataset and Pre-processing

For training the substitute model and for evaluating our approach, we use the preprocessed Java dataset used by the authors for Code2Vec (Alon et al., 2018). The dataset was collected from top-ranked Java repositories from GitHub. It has four versions - small, medium, 14m (with 14 million samples), and large (with 16 million samples). We use java-small for all of our experiments due to limited resources. The java-small database consists of

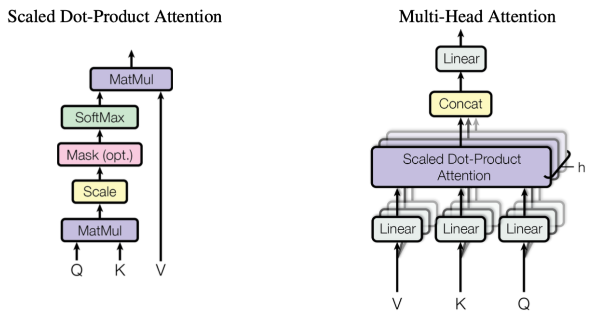


Figure 5: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel. (Vaswani et al., 2017)

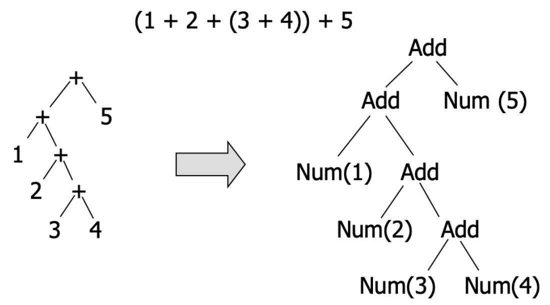


Figure 6: A toy example demonstrating Abstract Syntax Tree. Add is a non-terminal here. Num(...) are the terminals. Example taken from (Pingali, 2023).

9 Java projects for training, 1 for validation and 1 for testing and has about 700K code samples. The data is preprocessed for the model to be able to consume. The preprocessing utilizes the Abstract Syntax Tree (AST) of the code.

4.1 Abstract Syntax Tree

AST is used by compiler as an intermediate representation when compiling the code. As a very simple example to understand what an AST is, take a look at Figure 6.

Formally, an AST is defined as the tuple $< N, T, X, s, \delta, \phi >$ where N is the set of non-terminal nodes, T is the set of terminal nodes, X is the set of values, $s \in N$ is the root node, $\delta : N \rightarrow (N \cup T)^*$ maps a non terminal node to a list of its children, and $\phi : T \rightarrow X$ maps each terminal node to its value. Every node except the root occurs exactly one across all the lists of children.

An AST path is then defined as a path from one terminal node to another in the AST. At each point, the path can go either up or down the tree. For any AST path p of length k, $start(p)$ denotes the starting terminal of p, $end(p)$ denotes the ending terminal of p, and k denotes the number of edges on the path. Finally, the path context is then defined as the tuple $< x_s, p, x_t >$ where $x_s = \phi(start(p))$ and $x_t = \phi(end(p))$.

Given any code snippet C, it is processed to generate a representation that can be sent as the input to the model. First, the AST is generated using the language's parser, then for all pairs of terminal nodes t_i, t_j, we obtain the following representation as a bag of path contexts $Rep(C) = \{(x_s, p, x_t)\}$ such that the following holds:

$$x_s = \phi(t_i) \wedge x_t = \phi(t_j)$$

$$\wedge\, start(p) = t_i \wedge end(p) = t_j$$

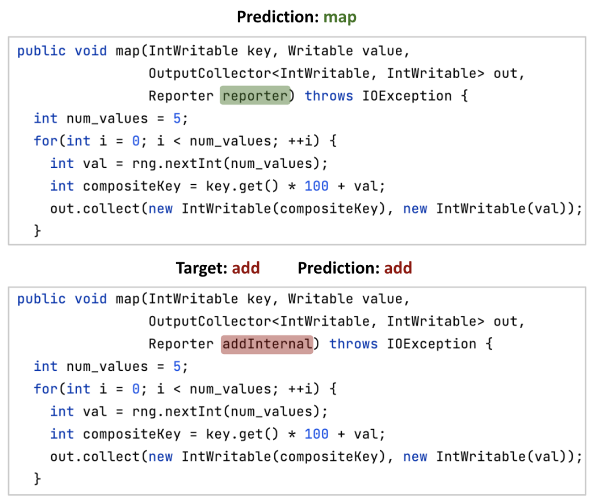
The context path length (k) and the width (maximum difference in child index between two children of the same intermediate node) can be controlled to limit the number of path contexts generated by the pre-processing. We used the same parameters as used by the authors in (Alon et al., 2018). Finally, the input to the model then is formed by the embedding for each such context path in $Rep(C)$ corresponding to code C.

5 Results

In this section, we describe the various experiments we ran to evaluate and understand the performance of our model. To start with, we trained the model described in Section 3 to mimic the Code2Vec (Alon et al., 2018) outputs by using its prediction outputs as the gold labels. The model achieved 68.83% top-1 accuracy and an F1 score of 69.18 in mimicing Code2Vec (Alon et al., 2018). The performance seems quite reasonable given that Code2Vec (Alon et al., 2018) itself has an F1 score of 59.5 on the sampled test set and 58.4 on the full test set, which speaks to the task's level of difficulty. Further experiments (described ahead) show that the substitute model still worked well for attacking the intended models.

5.1 Attacking Code2Vec

As planned, the first thing we tested out was if we could attack Code2Vec (Alon et al., 2018) without accessing its gradients. We used a black-box approach that limited us to using the model's output prediction on certain given inputs. In the following subsections, we provide and discuss some code snippets showing examples of the successful attacks using our substitute model.


Prediction: map

```
public void map(IntWritable key, Writable value,
                OutputCollector<IntWritable, IntWritable> out,
                Reporter reporter) throws IOException {
  int num_values = 5;
  for(int i = 0; i < num_values; ++i) {
    int val = rng.nextInt(num_values);
    int compositeKey = key.get() * 100 + val;
    out.collect(new IntWritable(compositeKey), new IntWritable(val));
  }
```

Target: add **Prediction:** add

```
public void map(IntWritable key, Writable value,
                OutputCollector<IntWritable, IntWritable> out,
                Reporter addInternal) throws IOException {
  int num_values = 5;
  for(int i = 0; i < num_values; ++i) {
    int val = rng.nextInt(num_values);
    int compositeKey = key.get() * 100 + val;
    out.collect(new IntWritable(compositeKey), new IntWritable(val));
  }
```

Figure 7: First code snippet shows an example of a correct prediction for the function map. The second Java snippet shows an example of the targeted attack output and prediction for the add, it correctly fools the model.

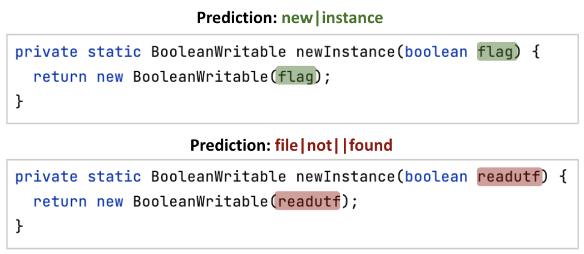
5.1.1 Targeted attacks

Fig. 7 shows a successful targeted black-box attack against Code2Vec (Alon et al., 2018). As can be seen, changing the variable name from "reporter" to "addInternal" fooled the model into prediction the function name "add" instead of the correct "map". An interesting finding in our experiments was that with most successful attacks for the target "add", the adversarial variable name usually contained the sub-token "add", which seems to be sufficient to trick the model. We include some more examples with different targets in Appendix A.

5.1.2 Untargeted attacks

Fig. 8 shows a successful untargeted black-box attack against Code2Vec (Alon et al., 2018). As seen in the sample, adversarially changing the name of variable 'flag" to "readutf" seems to trick the model into predicting "fileNotFound" as the function name instead of "newInstance". For a lot of successful attacks, there also seems to be some relation between the adversarial variable name and the predicted label. In this example as well, changing the variable name to "read**utf**" makes the model think some file processing is involved.

As we expected, the model was more successful with untargeted attacks than it was with targeted attacks, simply because an untargeted attack comparatively has an easier goal.


Prediction: new|instance

```
private static BooleanWritable newInstance(boolean flag) {
  return new BooleanWritable(flag);
}
```

Prediction: file|not||found

```
private static BooleanWritable newInstance(boolean readutf) {
  return new BooleanWritable(readutf);
}
```

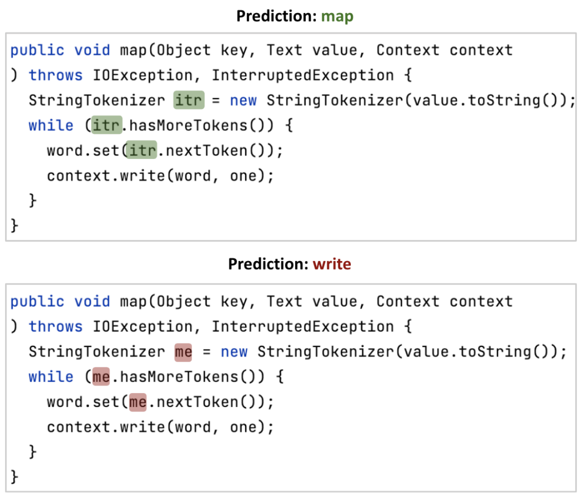
Figure 8: First code snippet shows an example of a correct prediction for the function *newInstance*. The second Java snippet shows an example of an untargeted attack prediction of *fileNotFound*, it correctly fools the model.

5.2 Testing the attack's generalizability

Next, to further test our approach's strengths and boundaries, we wanted to see if the attack could be generalized to other models. Particularly, we wanted to see if the model could be used to attack other models in a similar manner even though it only learned from Code2Vec (Alon et al., 2018). This does two things: test other models' robustness, and test the strength of the attack (if training a single model suffices for attacking multiple models, it makes it easier for an adversary to attack more models with low effort).

We tried our attack against two such models: Code2Seq (Alon et al., 2019) and ChatGPT (OpenAI, 2023). Figures 9 and 10 show successful attacks for the function map for these models. As can be seen in these examples, both models were fooled into the wrong prediction. What seems worse is that the models seem to be confused to the extent where they are just returning tokens from the code snippet as their response for the function name. This is much more apparent with ChatGPT (OpenAI, 2023) as it starts including punctuations (".") in the function name which results in an invalid function name. We include some more examples of successful attacks against ChatGPT in Appendix B demonstrating similar behavior.

Overall, we found that the attack generalizes decently well to other models as well. While not all code snippets that fooled Code2Vec (Alon et al., 2018) fooled Code2Seq (Alon et al., 2019) and ChatGPT (OpenAI, 2023), it still fooled them on a significant number of samples. Based on our understanding, we hypothesize that the substitute model understands the key areas of the code snippet, and modifies those to fool the model. While the code models may learn in a different manner, the key ar-



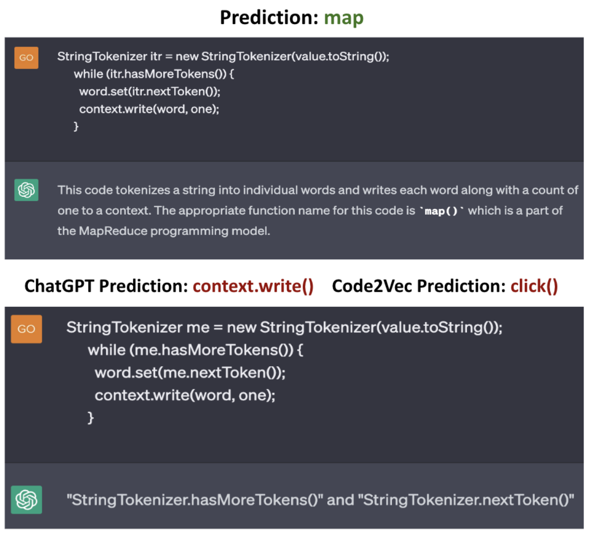
Prediction: map

```
public void map(Object key, Text value, Context context
) throws IOException, InterruptedException {
  StringTokenizer itr = new StringTokenizer(value.toString());
  while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    context.write(word, one);
  }
}
```

Prediction: write

```
public void map(Object key, Text value, Context context
) throws IOException, InterruptedException {
  StringTokenizer me = new StringTokenizer(value.toString());
  while (me.hasMoreTokens()) {
    word.set(me.nextToken());
    context.write(word, one);
  }
}
```

Figure 9: Example of incorrect prediction from the model code2seq for function map, and thus, a successful attack against code2seq.



Prediction: map

```
StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      context.write(word, one);
    }
```

This code tokenizes a string into individual words and writes each word along with a count of one to a context. The appropriate function name for this code is `map()` which is a part of the MapReduce programming model.

ChatGPT Prediction: context.write() Code2Vec Prediction: click()

```
StringTokenizer me = new StringTokenizer(value.toString());
    while (me.hasMoreTokens()) {
      word.set(me.nextToken());
      context.write(word, one);
    }
```

"StringTokenizer.hasMoreTokens()" and "StringTokenizer.nextToken()"

Figure 10: Example of incorrect prediction from ChatGPT for the function map, and thus, a successful attack against ChatGPT.

eas of the code would remain the same, and should impact most, if not all, models that utilize the information there. In the next section, we analyze the attention weights in the model for different paths to see if our hypothesis holds true.

5.3 Analyzing the attention

To understand how the model decides which variable name to change, we analyze the pattern of attention weights on various code snippets and see if there's a correlation between high attention paths and the variable name changed. Figure 11 shows the AST for the original code in Figure 9. On changing the variable "itr" to "me" fools Code2Vec (Alon et al., 2018) into predicting "click". The AST also shows the high attention paths - the width of each of the highlighted paths is proportional to the attention weight for the path. As we see here, "itr" lies on the path with the highest attention, which explains why changing it has a major impact on the model's prediction. Based on the empirical evaluation of high attention paths on some of the samples, it seems the substitute model usually changes some variable name on a high attention path to have a higher impact on the code model's prediction.

6 Discussion and Future Work

As mentioned in Section 5, we see that the code models are vulnerable to even black-box attacks. What was surprising was the success of the attack on ChatGPT (OpenAI, 2023) as well, which can be claimed to be the current state-of-the-art and has probably been trained on a lot more code, and also gathers its knowledge from non-code sources as well. The success of such an attack is concerning for multiple reasons. First and foremost, this raises doubts on all responses generated by these models, and brings into question what exactly it is that these models are learning and thus, raises questions on their reliability and trustworthiness.
But even more concerning is the ease of the attack. While the substitute model was trained using Code2Vec (Alon et al., 2018), and Code2Seq (Alon et al., 2019) is also trained on the same dataset, we can at least assume that it has "some" information about these models that it may be leveraging in the attack. But the same attack being successful against ChatGPT (OpenAI, 2023), without any knowledge of its architecture, or what data it has been trained on, indicates that its much easier to fool these models. This demonstrates the importance of robust

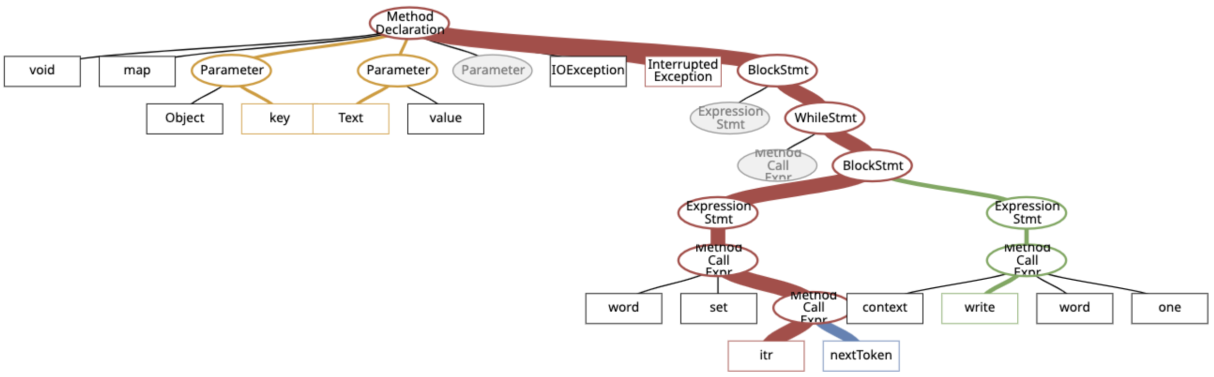
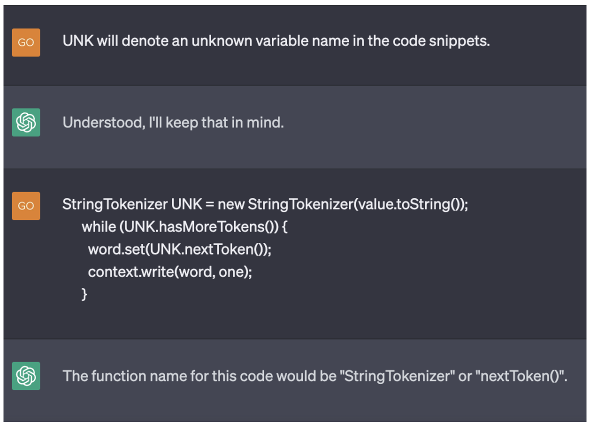


Figure 11: Attention paths for code snippet "map" shown in original code of Figure 9. Here we can see that the variable "itr" is contained in the path with the highest attention. (Alon et al., 2018)

models and that robustness should be considered as important a metric as accuracy. It also brings to forefront the need to good defenses against such attacks. One such defense that (Yefet et al., 2019) mentions is to strip the code of variable name information but only at test time, which the authors claim should be 100% robust against this attack. But if the model has already dependent on the variable names, stripping the code of this information might form an attack in itself, albeit untargeted, leading the model to an incorrect prediction. Figure 12 shows an example where ChatGPT was still confused into the wrong prediction after stripping the code of adversarial variable name.
On a similar note, it is also concerning that the same model can be used to attack multiple models of code. This would make any adversary's job much easier, and essentially lead to models being more vulnerable.
Additionally, as future work, it may be interesting to explore a couple things: first, it might be interesting to see the impact of dataset used to train the substitute model. Currently, for training the substitute model, we used a subset of the dataset used to train the original Code2Vec (Alon et al., 2018) model. It might be easier for the substitute model to mimic the original model when it is trained on its outputs on the same dataset, but this knowledge may not always be available. So, training it on another dataset might reveal if a good substitute model can be trained even without the knowledge of the dataset.
Another potentially interesting exploration could be to test the models' dependence on the variable names. Since what we're manipulating in this attack are the variable names, it clearly indicates that the models depend quite a bit on the variable names



UNK will denote an unknown variable name in the code snippets.

Understood, I'll keep that in mind.

```
StringTokenizer UNK = new StringTokenizer(value.toString());
   while (UNK.hasMoreTokens()) {
     word.set(UNK.nextToken());
     context.write(word, one);
   }
```

The function name for this code would be "StringTokenizer" or "nextToken()".

Figure 12: Example of ChatGPT still being fooled after removing the adversarial variable name.

to generate their predictions. It should be interesting to see how these models work if we strip all code of variable name information - say, by either using the variable names from a minified version of the code, or by using dummy variable names (say, they're chosen lexically over all possible variable names, starting from "a"). Another interesting experiment could be to train a similar model like Code2Vec (or any of the vulnerable models), but by hiding the variable name information from the dataset. Since this should strip the model of its dependence on variable names, it should ideally make it robust against this attack, but it might be harder to achieve a good accuracy without the information about variable names.

7 Challenges

One of our biggest challenge during the project was working with TensorFlow 1. The code for (Yefet et al., 2019) that we used as our starting point was written in TensorFlow 1. TensorFlow

1 is not supported on Google Colaboratory either, which restricted us to TACC even for small experiments. Additionally, since it is fundamentally different from the latest version (TensorFlow 2) in a lot of aspects, it was also hard finding sufficient resources, particularly considering our lack of prior experience with TensorFlow. Making the appropriate changes in the code to include a substitute model took a much longer time than we expected because of this. As with most machine learning based projects, we also were constrained on time, which restricted us to experiments with the small dataset only, thus limiting the performance of the substitute model as well.

References

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. code2vec: Learning distributed representations of code.

Siddhant Bhambri, Sumanyu Muku, Avinash Tulasi, and Arun Balaji Buduru. 2020. A survey of black-box adversarial attacks on computer vision models.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages.

Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and harnessing adversarial examples.

Akshita Jha and Chandan K. Reddy. 2022. Codeattack: Code-based adversarial attacks for pre-trained programming language models.

OpenAI. 2023. Gpt-4 technical report.

Keshav K Pingali. 2023. Cs380c compilers lecture slides.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation.

Noam Yefet, Uri Alon, and Eran Yahav. 2019. Adversarial examples for models of code.

Wei Emma Zhang, Quan Z. Sheng, Ahoud Alhazmi, and Chenliang Li. 2019. Adversarial attacks on deep learning models in natural language processing: A survey.

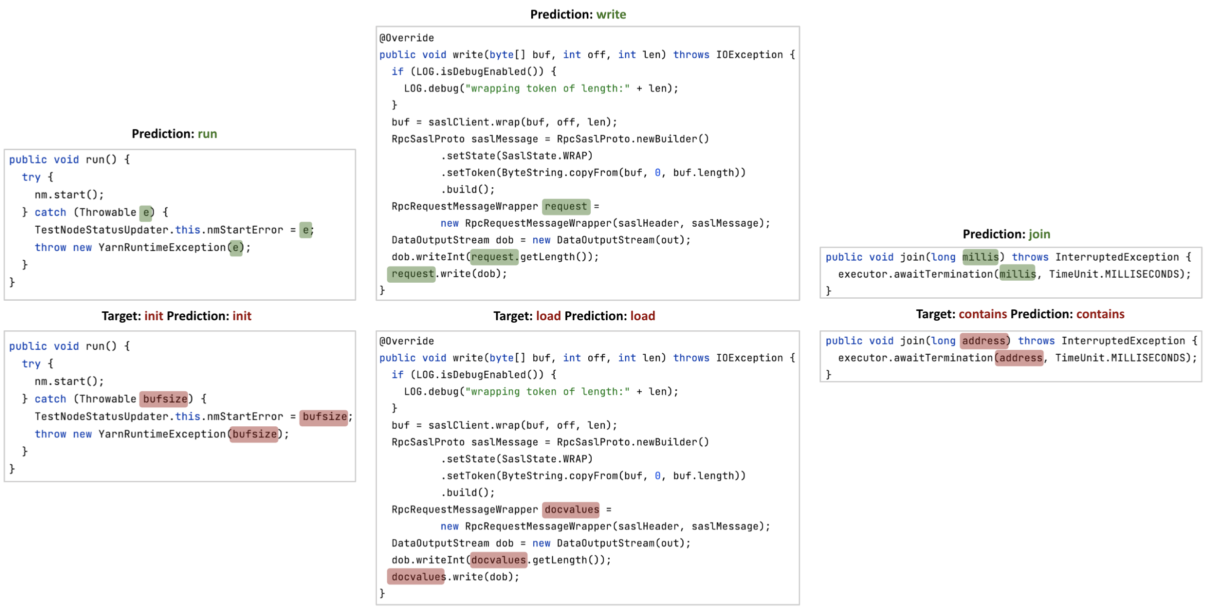
Yu Zhou, Xiaoqing Zhang, Juanjuan Shen, Tingting Han, Taolue Chen, and Harald Gall. 2021. Adversarial robustness of deep code comment generation.

A Samples of Targeted Attacks

Figure 13 shows some additional examples of successful attack against Code2Vec (Alon et al., 2018) using the substitute model. In all cases here as well, we see some relation between the adversarial variable name and the target adversarial label.

B Samples of Adversarial Attacks for ChatGPT

Figure 14 shows some more examples of ChatGPT (OpenAI, 2023) being fooled by the substitute model's adversarial attack. In the examples here, it again seems to just be returning different tokens from the code snippet instead of understanding what the code does.



Prediction: run

```
public void run() {
  try {
    nm.start();
  } catch (Throwable e) {
    TestNodeStatusUpdater.this.nmStartError = e;
    throw new YarnRuntimeException(e);
  }
}
```

Target: init Prediction: init

```
public void run() {
  try {
    nm.start();
  } catch (Throwable bufsize) {
    TestNodeStatusUpdater.this.nmStartError = bufsize;
    throw new YarnRuntimeException(bufsize);
  }
}
```

Prediction: write

```
@Override
public void write(byte[] buf, int off, int len) throws IOException {
  if (LOG.isDebugEnabled()) {
    LOG.debug("wrapping token of length:" + len);
  }
  buf = saslClient.wrap(buf, off, len);
  RpcSaslProto saslMessage = RpcSaslProto.newBuilder()
          .setState(SaslState.WRAP)
          .setToken(ByteString.copyFrom(buf, 0, buf.length))
          .build();
  RpcRequestMessageWrapper request =
          new RpcRequestMessageWrapper(saslHeader, saslMessage);
  DataOutputStream dob = new DataOutputStream(out);
  dob.writeInt(request.getLength());
  request.write(dob);
}
```

Target: load Prediction: load

```
@Override
public void write(byte[] buf, int off, int len) throws IOException {
  if (LOG.isDebugEnabled()) {
    LOG.debug("wrapping token of length:" + len);
  }
  buf = saslClient.wrap(buf, off, len);
  RpcSaslProto saslMessage = RpcSaslProto.newBuilder()
          .setState(SaslState.WRAP)
          .setToken(ByteString.copyFrom(buf, 0, buf.length))
          .build();
  RpcRequestMessageWrapper docvalues =
          new RpcRequestMessageWrapper(saslHeader, saslMessage);
  DataOutputStream dob = new DataOutputStream(out);
  dob.writeInt(docvalues.getLength());
  docvalues.write(dob);
}
```

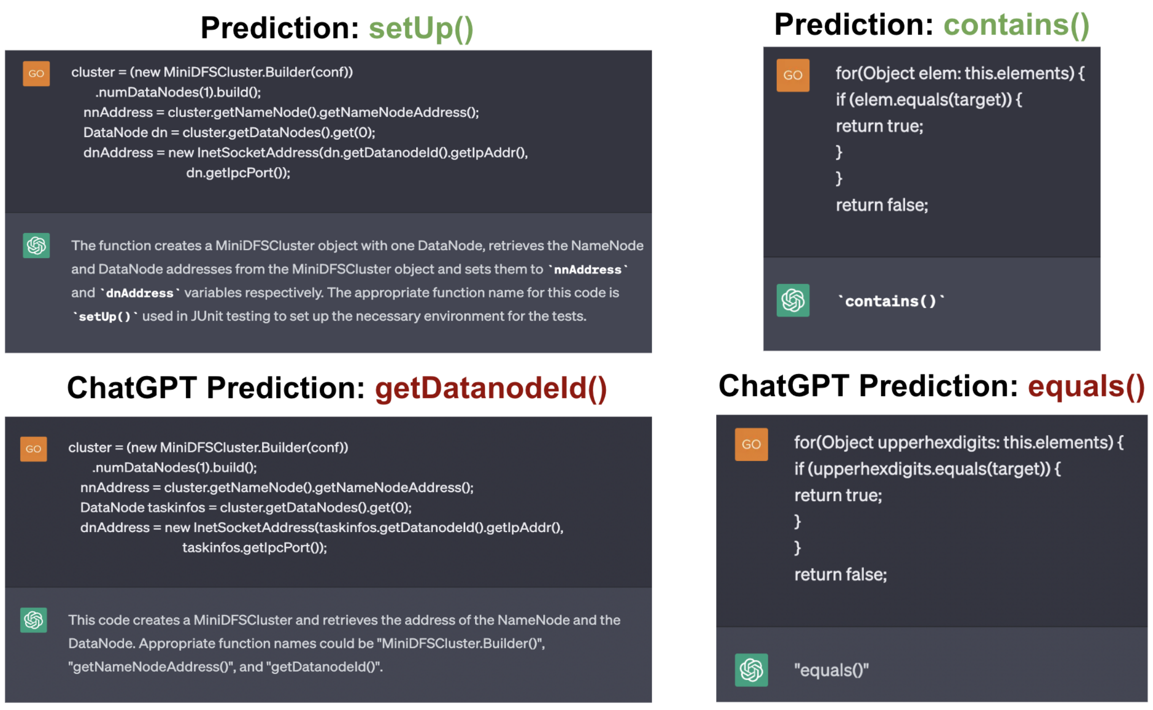
Prediction: join

```
public void join(long millis) throws InterruptedException {
  executor.awaitTermination(millis, TimeUnit.MILLISECONDS);
}
```

Target: contains Prediction: contains

```
public void join(long address) throws InterruptedException {
  executor.awaitTermination(address, TimeUnit.MILLISECONDS);
}
```

Figure 13: Examples of successful targeted attacks against Code2Vec.



Prediction: setUp()

```
cluster = (new MiniDFSCluster.Builder(conf))
    .numDataNodes(1).build();
  nnAddress = cluster.getNameNode().getNameNodeAddress();
  DataNode dn = cluster.getDataNodes().get(0);
  dnAddress = new InetSocketAddress(dn.getDatanodeId().getIpAddr(),
                dn.getIpcPort());
```

The function creates a MiniDFSCluster object with one DataNode, retrieves the NameNode and DataNode addresses from the MiniDFSCluster object and sets them to `nnAddress` and `dnAddress` variables respectively. The appropriate function name for this code is `setUp()` used in JUnit testing to set up the necessary environment for the tests.

ChatGPT Prediction: getDatanodeId()

```
cluster = (new MiniDFSCluster.Builder(conf))
    .numDataNodes(1).build();
  nnAddress = cluster.getNameNode().getNameNodeAddress();
  DataNode taskinfos = cluster.getDataNodes().get(0);
  dnAddress = new InetSocketAddress(taskinfos.getDatanodeId().getIpAddr(),
                taskinfos.getIpcPort());
```

This code creates a MiniDFSCluster and retrieves the address of the NameNode and the DataNode. Appropriate function names could be "MiniDFSCluster.Builder()", "getNameNodeAddress()", and "getDatanodeId()".

(a)

Prediction: contains()

```
for(Object elem: this.elements) {
if (elem.equals(target)) {
return true;
}
}
return false;
```

`contains()`

ChatGPT Prediction: equals()

```
for(Object upperhexdigits: this.elements) {
if (upperhexdigits.equals(target)) {
return true;
}
}
return false;
```

"equals()"

(b)

Figure 14: Examples of successful attacks against ChatGPT