Deceiving Neural Source Code Classifiers: Finding Adversarial Examples with Grammatical Evolution

Claudio Ferretti
claudio.ferretti@unimib.it
Dept. of Informatics, Systems and Communication,
University of Milano-Bicocca
Milan, Italy

Martina Saletta
martina.saletta@unimib.it
Dept. of Informatics, Systems and Communication,
University of Milano-Bicocca
Milan, Italy

ABSTRACT

This work presents an evolutionary approach for assessing the robustness of a system trained in the detection of software vulnerabilities. By applying a Grammatical Evolution genetic algorithm, and using the output of the system being assessed as the fitness function, we show how we can easily change the classification decision (i.e. vulnerable or not vulnerable) for a given instance by simply injecting evolved features that in no wise affect the functionality of the program. Additionally, by means of the same technique, that is by simply modifying the program instances, we show how we can significantly decrease the accuracy measure of the whole system on the dataset used for the test phase.

Finally we remark that these methods can be easily customized for applications in different domains and also how the underlying ideas can be exploited for different purposes, such as the exploration of the behaviour of a generic neural system.

CCS CONCEPTS

• **Security and privacy** → **Software and application security**; • **Computing methodologies** → *Genetic algorithms*; *Neural networks*.

KEYWORDS

grammatical evolution, deep learning, adversarial examples, computer security, security assessment

ACM Reference Format:
Claudio Ferretti and Martina Saletta. 2021. Deceiving Neural Source Code Classifiers: Finding Adversarial Examples with Grammatical Evolution. In *2021 Genetic and Evolutionary Computation Conference Companion (GECCO '21 Companion), July 10–14, 2021, Lille, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3449726.3463222

1 INTRODUCTION

Statistical classification, namely the problem of assigning, given a set of instances and a set of categories, an instance to the correct category, has been widely studied in the years, especially with the diffusion of machine learning methods since the '80s [17].

Several examples exist for illustrating the significance of classification algorithms in a wide variety of domains: it suffices to think to natural language processing (NLP) techniques for sentiment analysis [21], to tasks related to image classification [6] or, in the wider sense, to the methods for grouping the elements of a set according to their similarity (e.g. cluster analysis [8]). The interest is also in the field of cybersecurity: as simple examples, phishing and spam detectors [14] but also malware detectors [13] are all evidences of the importance of being able to effectively distinguish between what is safe to what it is not, and thus to classify objects such as emails, files and programs.

However, binary classifiers are often susceptible to classification errors. While in some cases such misclassifications only affect the accuracy measure, in some scopes the presence of false positive or false negative is crucial; just think, as an example, to antivirus: in this context, a false positive malware alert it is not a big deal, while a false negative could be critical. For this reason, evaluating the robustness of a classifier is essential. If a classifier system is widespread, it is possible that an attacker looks for features of input source code that make it evade the surveillance. The system has to be evaluated with respect to such risks.

When this task is related to image classification, where instances are vectors of values from a continuous domain (pixel colors), the approach is to check the results of small variations of input values. The challenge is harder when instances are source code fragments, where each variation on their feature values is actually a variation of a syntactic (sub)tree, which must obey the grammar of the chosen programming language.

In this work, we are interested in designing an evolutionary approach for assessing the robustness of a classification system by producing program instances that mislead a system trained in the detection of software vulnerabilities. In other words, our goal is to synthesise vulnerable programs that will be classified as safe by the system, or vice versa. To this end, this work offers the following contributions:

- A novel evolutionary approach for synthesising programs using, as the fitness function, the output of a neuron in a neural model.
- The application of this approach in the construction, or adaptation, of a dataset so that it leads to a high percentage of false negative (or, possibly, false positive) misclassifications on a network trained in the detection of software vulnerabilities.
- The use of the best individuals for characterising the syntactical features that mainly stimulate a given neuron.

- How such characterization can be used for modifying the input instances of a classifier in order to arbitrarily change the classification decision.

The rest of the paper is organized as follows:

Section 2 is a short overview of how the topics considered are approached in this work. In particular, it discusses how machine learning techniques have been exploited for detecting and classifying software vulnerabilities, it describes how Grammatical Evolution (GE) can be employed for program synthesis and it briefly presents adversarial machine learning approaches and explainability in neural networks.

Section 3 describes the approach proposed, namely the application of GE in the synthesis of programs aiming to deceive a neural network trained in vulnerability detection, and introduces some terminology used in the paper.

Section 4 presents the experimental settings, the adopted libraries, tools and methods, and details all the experiments we designed and performed. It finally reports the outcomes obtained.

Section 5 discusses the results and gives intuition of their interest and relevance. In particular, we examine how the wide range probing of the classifier behavior via GE-evolved individuals allows us to discover its weaknesses and to use them for deceiving it.

Section 6 finally provides a summary of the ideas contained in this paper and suggests how the insights yielded by our results can be transferred and adopted in several different domains.

All the results described in this paper, along with the source code, are available in a public repository[1].

2 RELATED WORK

In this work, we present an evolutionary-based pipeline for program synthesis, with the aim to assess the robustness of a classifier trained in the detection of software vulnerabilities, given the source code. This section provides a brief literature overview of the main topics involved in our discussion, focusing the attention on the perspective we are interested in.

2.1 Machine learning for vulnerability detection

Classifying programs or, more in general, the problem of automatically identifying a correct tag or property for a snippet of source code is significant in several fields, and it is quite often addressed by the use of machine learning techniques. For instance, the authors of [22] propose a Convolutional Neural Network (CNN) for classifying programs according to their functionality, while other works strive to predict a word or a short sentence that summarizes the purpose of the code using different techniques such as a CNN based on the attention model [2] or a source code embedding derived from the abstract syntax tree [3].

In the field of cybersecurity, aside from the classical static and dynamic analysis techniques for detecting malware [13] or vulnerabilities [20], in this context we are particularly interested in vulnerability detection with machine learning approaches. To this end, many works can be mentioned: for instance, in [32] a program embedding and common usage patterns are used for identifying functions vulnerable to known weaknesses, the authors of [9] propose a Long Short Term Memory network for detecting vulnerabilities in PHP programs, while [19] is a complete deep learning-based framework for the identifying weaknesses in C programs.

In the rest of this paper, we will particularly refer to the network for vulnerability detection proposed in [28], since our experiments are focused on producing adversarial examples for that model, through the design of a GE-based pipeline that can be easily adapted to different neural models.

2.2 Program synthesis with GE

Grammatical Evolution [25] is an evolutionary algorithm that, similarly to what Genetic Programming (GP) [16] does with syntax trees, can evolve programs (represented through a binary string) that comply with a given formal grammar expressed in Backus-Naur Form (BNF). For its inherent nature, GE can be applied for addressing the problem of program synthesis [5]. Although a recent work [30] illustrates some of its limitations in effectively solving general related tasks, other works point out its effectiveness under certain conditions. In particular, since the process of synthesising a program is evidently connected to the features of the specific programming problem to be solved, the authors of [12] show how the knowledge of the problem domain can be used for designing a grammar that effectively features productions that enable actions useful towards the solution of the problem itself. A similar intuition is exploited in [24], where GE is used for synthesising programs that solve the classical problem of integer sorting.

Starting from these results, we propose a GE algorithm that can be used for producing programs to be used for misleading a classifier, by proposing a simplified C grammar (Figure 8) with only a small subset of relevant functions included in the productions.

2.3 Adversarial examples and explainability

Any machine learning result obtained by using deep neural networks has one limit: it suffers the fact that the neural network is mostly a black box, with respect to the explainability of how it is producing its output and whether it hides problematic decision points. Evidence of this is discussed in works related to *adversarial examples*, which are input instances crafted to fool state-of-the-art neural networks. In addition, another key element is that such adversarial instances can also be only slightly different from instances which instead are correctly classified by the same machine.

This is especially apparent in the field of image recognition [11], where some research results show how to find images which are strong adversarial examples, usually by examining the gradient of the cost function of the given backpropagation network [11]. Similar adversarial instances can be constructed for models built for source code processing, simply by applying basic semantic-preserving perturbations to the source code, for instance by renaming some identifiers or by introducing unused variables [33, 34], or more sophisticated ones, such as by using different control flow structures or by changing the API usage [27].

[1]https://github.com/Martisal/adversarialGE

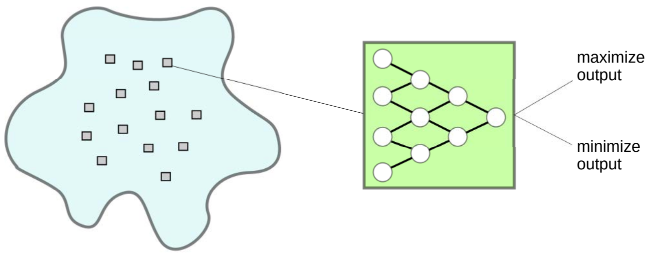
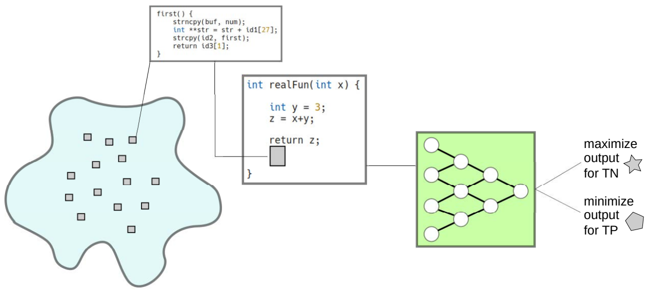


Figure 1: First class of experiments. Fitness function is the output of the network given the pure evolved individual as input.



```
first() {
    strncpy(buf, num);
    int **str = str + id1[27];
    strcpy(id2, first);
    return id3[1];
}
```

```
int realFun(int x) {

    int y = 3;
    z = x+y;

    return z;

}
```

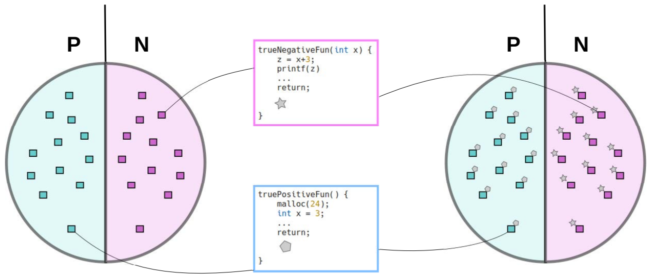
Figure 2: Second class of experiments. Fitness function is the output of the network given an hybrid input obtained by injecting an evolved individual in a true positive (if minimising the fitness) or a true negative (if maximising the fitness) instance of the original dataset.

Getting back to the image processing field, in [23] the adversarial examples are generated both by gradient ascent and by using evolutionary algorithms. In our work we apply a similar evolutionary approach for the domain of source code classification instead of that of images.

When looking for adversarial examples, new knowledge is gathered about the behavior of the network, and this knowledge helps us in the explanation of what is happening inside it [7]. Given the increasing role of machine learning systems in many decisions impacting the society, or just the security of many modern systems, the explainability of artificial intelligence tools is a growing research area [1, 31]. We will make some remarks on how our results relate to the topics of assessment and explainability of trained classifiers.

3 PROPOSED APPROACH

In this paper, we propose an evolutionary approach for assessing the robustness of a model trained in the detection of software vulnerabilities. The founding idea is to synthesise input instances able to lead the network to a wrong classification decision, by exploring the space of possible solutions using a GE algorithm, with the numerical output of the model as the fitness function. This insight allows us to efficiently explore the behaviour of a classifier by leveraging an evolutionary pressure instead of randomly sampling the solution space, and thus to possibly discover some blind spots or some features that can mislead the model.



```
trueNegativeFun(int x) {
    z = x+3;
    printf(z)
    ...
    return;

}
```

```
truePositiveFun() {
    malloc(24);
    int x = 3;
    ...
    return;

}
```

Figure 3: Classification experiments. The original dataset (on the left) is modified to generate a new dataset (on the right) by injecting, in all the instances, an evolved individual.

The authors of [28] propose a tool for detecting software vulnerabilities using a deep feature representation based on a lexical tokenization of the source code. In that work, an approach derived from classical NLP techniques for sentiment analysis [15] is developed for classifying programs which are potentially vulnerable to known categories of software weaknesses (CWE[2]). Using this model as a benchmark, we designed three classes of experiments:

- The evolution of *pure* individuals able to maximize or minimize the output of the network, as summarized in Figure 1. In these experiments, the fitness function is computed as the output of the network given the evolved individual as the input.
- The injection of evolved individuals in functions of the original dataset, in a way that does not affect the behaviour of the function (i.e. after the return statement, as will be explained in Section 4). In this case, the fitness function is computed as the output of the network given as input the *hybrid* program consisting of the original functions modified with the injection, as outlined in Figure 2. The aim of these experiments is to arbitrarily change the classification decision for a given instance, namely to maximise the fitness if starting from functions classified as negative and vice versa.
- The injection of evolved individuals in all the original test set, in order to change the classification statistics of the network (Figure 3). The goal is to assess our approach by analysing how accuracy, precision and recall change depending on how and which evolved individuals are used for the injection.

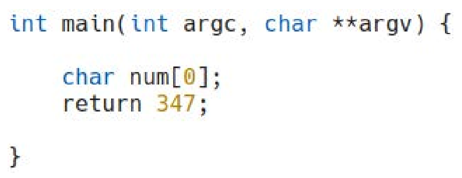
For ease of reference, we summarize below some terminology that will be used in the rest the paper.

Individual An evolved program $\mathcal{P}$. It consists of a snippet of code that complies with the formal grammar specified in Figure 8. Examples can be found in Figures 4 and 5.

Pure individual An evolved program $\mathcal{P}$ whose fitness function $F : \mathcal{P} \mapsto [0, 1]$ is computed as the output of a neural model given $\mathcal{P}$ as input.

Hybrid individual An evolved program $\mathcal{P}$ whose fitness function $F : \mathcal{P} \mapsto [0, 1]$ is computed as the output of a neural model given as input an instance $\mathcal{P}_0$ from the dataset after the semantic-preserving injection of $\mathcal{P}$.

[2]https://cwe.mitre.org/

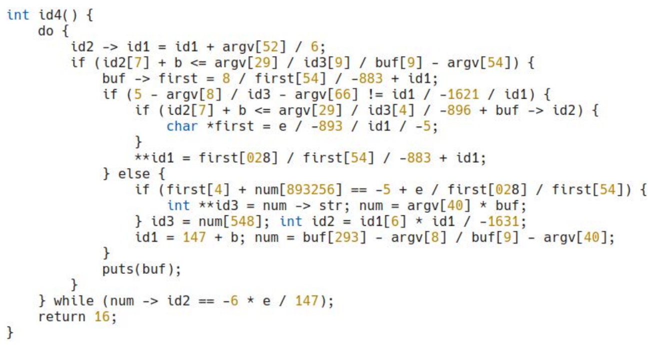


```
int main(int argc, char **argv) {

    char num[0];
    return 347;

}
```

Figure 4: Example of an evolved individual, maximised fitness function.



```
int id4() {
    do {
        id2 -> id1 = id1 + argv[52] / 6;
        if (id2[7] + b <= argv[29] / id3[9] / buf[9] - argv[54]) {
            buf -> first = 8 / first[54] / -883 + id1;
            if (5 - argv[8] / id3 - argv[66] != id1 / -1621 / id1) {
                if (id2[7] + b <= argv[29] / id3[4] / -896 + buf -> id2) {
                    char *first = e / -893 / id1 / -5;
                }
                **id1 = first[028] / first[54] / -883 + id1;
            } else {
                if (first[4] + num[893256] == -5 + e / first[028] / first[54]) {
                    int **id3 = num -> str; num = argv[40] * buf;
                } id3 = num[548]; int id2 = id1[6] * id1 / -1631;
                id1 = 147 + b; num = buf[293] - argv[8] / buf[9] - argv[40];
            }
            puts(buf);
        }
    } while (num -> id2 == -6 * e / 147);
    return 16;
}
```

Figure 5: Example of an evolved individual, minimised fitness function.

Semantic-preserving injection The addition of instructions in a program that do not alter the *semantic* of the program itself. In this paper, we consider only the following transformations: given two programs $\mathcal{P}_0$ and $\mathcal{P}$, we call the semantic-preserving injection of $\mathcal{P}$ in $\mathcal{P}_0$ the addition of the instructions contained in $\mathcal{P}$ after the return statement of $\mathcal{P}_0$.

Positive (or negative) instance We refer to the vulnerable (or safe) programs in the dataset as positive (or negative) instances. It will be clear in turn if positive (or negative) refers to the ground truth or to the classification decision of the model.

4 EXPERIMENTS

We organized experiments along two directions: the use of GE to evolve source code fragments, forcing their binary classification by a given trained neural network, and to check how selected evolved individuals could be good to alter the classification of any instance from a labeled dataset.

The goal is to show how easily an attacker can mask the vulnerabilities in any source code instance, and thus to foil the classification by a neural network, even if treated as a black box.

As a benchmark, we considered the deep learning classifier presented in [28], working with a 94% accuracy after training on the dataset curated by the same research group[3], and all our experiments have been performed on a test section of the same dataset, not used during the training phase.

[3]available at https://osf.io/d45bw/

4.1 Experimental Settings

The neural classifier is based on a preprocessing of source code where each instance, i.e. a C language function, is tokenized by a lexer, and then input to a deep stack of feedforward layers, including a 1-dimensional convolution, terminating in a single output value ranging from 0 to 1. The classification of the instance is positive (vulnerable) when the output value is higher than 0.5, negative otherwise. In particular, we trained the network on the supervised task of recognising vulnerabilities of type CWE-120 (classic buffer overflow).

The dataset, consisting in over one million labeled C functions, has been split in three folds, with ratio 80/10/10, used respectively for training/validation/testing. We remark that each of these folds has a number of instances strongly unbalanced between the two classes. During training, the number of instances of the two classes has been kept in a ratio of 5 negative instances for each positive instance.

The trained model exhibits a good accuracy of 94% on the test set, and among the available performance indices we will consider the precision/recall area under curve (P/R-AUC), more suitable given the unbalanced set of instances being used. The trained network has a P/R-AUC index of 0.42 on the chosen test set.

All the experiments have been done on a Linux machine with 16GB RAM and a Nvidia GTX 1070 GPU. The training phase took around 3 hours.

4.2 Grammatical Evolution

The library used for doing grammatical evolution of individuals consisting of C language functions is PonyGE2 [10]. The grammar, in Figure 8, has been derived from a realistic C language grammar by reducing the set of productions mainly to: declaration and use of integer, char, and buffer variables, single function declaration and call of functions from a small set of standard C functions. The choices controlling all evolutionary runs were: generational replacement, with elite of size 1, selection by tournament of size 3, with mutation and one-point crossover. All runs evolved for 33 generations, and individuals were of tree depth limited to 30.

In particular, we could specify to compute the fitness of individuals by preprocessing each of them as required by the neural classifier (by using the same tokenizing lexer built during training), and then reading the real valued classifying output of the network. Therefore, the fitness guiding the evolution of individuals was a value between 0 an 1, where values above 0.5 meant that the individual was being classified as vulnerable (w.r.t. CWE-120, in our case).

We remark that the tokenization considered only the 10000 most frequent words from the dataset, including C identifiers, operators, keywords and punctuation. The grammar we used to generate GE individuals (Figure 8) was built to produce both known (tokenized) identifiers as well as new ones, which are ignored by the tokenizer. For instance, the non-terminal `identifier` in our grammar includes both `buf`, known to the tokenizer defined during training, and `id1` which is not. This design choice is made in order to allow the GE to sample the input space with more freedom, so that adversarial examples could go beyond what the training set offered as dictionary.

Table 1: Summary of the statistics obtained in the classification experiments. *Minimized* and *Maximized* rows refer to the dataset modified by injecting one of the minimising and maximising evolved individuals, respectively; the *PureMaxMin* row refers to the dataset modified by injecting a random maximised pure individual in the negative instances and a random minimised pure individual in the positive instances; finally *HybrMaxMin* refers to the dataset where instances have been injected with a random hybrid individual, chosen matching the label of the instance as above. Last column reports the p-value obtained by performing the Mann-Whitney U Test on a sample from the dataset specified in the first column and a sample from the original dataset.

Dataset	Accuracy	P/R-AUC	F1	p-value
Original	0.94	0.42	0.51	(≈ 0.29)
Minimized	0.95	0.29	0.41	$\approx 10^{-3}$
Maximized	0.17	0.06	0.08	$\approx 10^{-29}$
PureMaxMin	0.67	0.06	0.1	$\approx 10^{-15}$
HybrMaxMin	0.18	0.02	0.03	$\approx 10^{-26}$

Table 2: Confusion matrix of the classification results obtained on the original test set (*Original* in Table 1)

Truth	Class. Negative	Class. Positive
Negative	116608	5920
Positive	1172	3719

Table 3: Confusion matrix of the classification results obtained on the test set modified by injecting, in each instance, one of the *minimizing* individuals evolved through the second class of experiments (see Figure 2, and row *Minimized* in Table 1).

Truth	Class. Negative	Class. Positive
Negative	118946	3582
Positive	2683	2208

Each evolutionary run was processed on the same machine we used for the training phase, and took less than 1 minute.

4.3 Results of GE runs

Given the two experimental platforms we described, a GE system to evolve C language functions, and a deep network trained to recognize buffer overflow vulnerabilities, we present the set of experiments we performed to assess the weaknesses of the neural classifier.

First, we verified how the GE system performed when required to define C functions classified as vulnerable (or non vulnerable), that is maximising the fitness function (or minimizing, respectively), with its value defined by inferring the classification of each individual of the evolving population. Even with the simplified grammar we used, both goals (neural output value above or below 0.5) have been easily reached on each evolutionary run.

An interesting outcome is that C functions maximising the fitness have been consistently short (less than 100 characters), while the individuals minimising it were always much longer (see examples in Figures 4 and 5).

We will refer to the individual produced in this first example as *pure*, being evolved with no reference to C functions from the dataset, and considered as *positive* when maximising the fitness, and as *negative* in the opposite case.

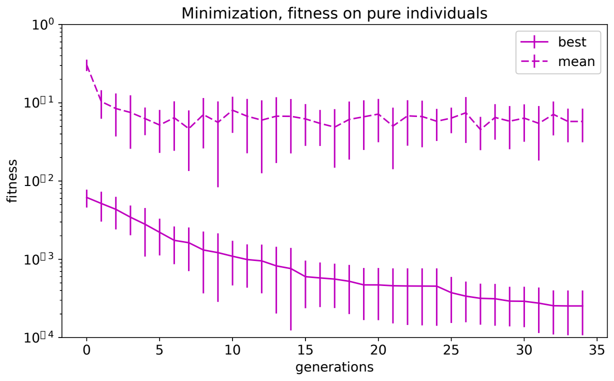
The second set of experiments has been aimed at evolving, by GE, C fragments to be injected in existing instances. We chose from the dataset some instances labeled as vulnerable and some not vulnerable. Then, starting from single given instance, we performed evolutionary runs where each GE individual was inserted in it, and the fitness was evaluated on the resulting source code of the function (see Figure 2). The individual was inserted in a way not to change the actual behavior of the given function, namely by appending it inside the instance right before the closing bracket, after any return, effectively becoming *dead code* with no effect at compile or run time. Other simple semantics-preserving transformations could also be used, such as blocks controlled by a condition which will never be true. The same has been done both for positive and for negative instances, and in the following we will refer to those modified instances as *hybrid* individuals.

In this way, we looked for a way to mask a possible vulnerability in a function to be classified, or conversely to force a positive classification of a function actually non vulnerable. Every experiment in this set has been successful, always finding an individual which, when injected in a given labeled function, made that same function being wrongly classified by the neural network. We observed that with this setting the winning individuals were more complex than those evolved as pure, as was expected, given the task to mask some vulnerability properties of the given enclosing function. The progress towards individuals with the wanted fitness (positive or negative) was still quite fast during each GE runs (compare Figures 6 and 7).

4.4 Results on the Classifier

We used the set of C source code fragments produced by GE, pure and hybrid individuals, to finally assess the robustness of the trained neural network in recognizing vulnerabilities in known instances.

The first check has been done by injecting pure individuals in known instances, with these obvious cautions: from the evolved functions we extracted only the body, and we inserted it in the instances so to obtain a function which was different, but syntactically correct and with the same original behavior (as explained before). The chosen pure individual was always generated with a



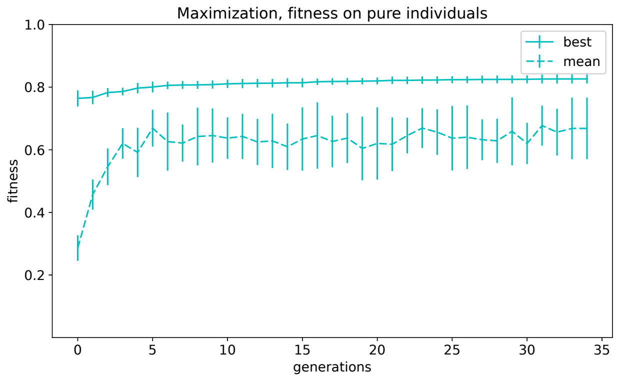
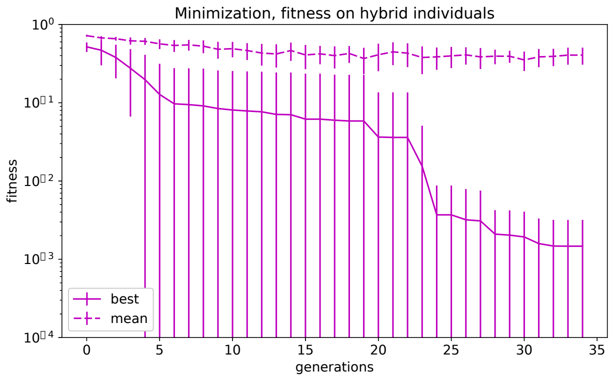


Figure 6: Fitness of the pure individuals evolved in the first class of experiments (Figure 1). Plots are built by considering the mean of the fitness values obtained over 10 runs, with vertical bars reporting the standard deviation. Dashed lines indicate the average fitness of the population at each generation, while continuous lines indicate the fitness of the best individual at each generation. Notice that, for the minimization experiments, plots are in logarithmic scale.



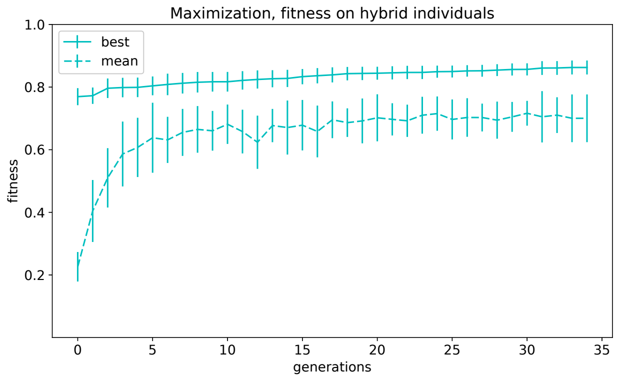


Figure 7: Fitness of the hybrid individuals evolved in the second class of experiments (Figure 2). Plots are built by considering the mean of the fitness values obtained over 10 runs, with vertical bars reporting the standard deviation. Dashed lines indicate the average fitness of the population at each generation, while continuous lines indicate the fitness of the best individual at each generation. Notice that, for the minimization experiments, plots are in logarithmic scale.

Table 4: Confusion matrix of the classification results obtained on the test set modified by injecting, in each instance, one of the *maximising* individuals evolved through the second class of experiments (see Figure 2, and row *Maximized* in Table 1).

Truth	Class. Negative	Class. Positive
Negative	16585	105943
Positive	43	4848

target fitness corresponding to a classification which was opposite to the true classification of the given instance.

Table 5: Confusion matrix of the classification results obtained on the test set modified by injecting, in each *positive* instance, a random pure *minimised* individual and, in each *negative* instance, a random *maximised* pure individual. Minimised and maximised individuals are randomly chosen among the best pure individuals found in different runs of the first class of experiments (see Figure 1 and Figure 3, and row *PureMaxMin* in Table 1).

Truth	Class. Negative	Class. Positive
Negative	83185	39343
Positive	2394	2497

Table 6: Confusion matrix of the classification results obtained on the test set modified by injecting, in each *positive* instance, a random hybrid *minimised* individual and, in each *negative* instance, a random *maximised* hybrid individual. Minimised and maximised individuals are randomly chosen among the best hybrid individuals found in different runs of the second class of experiments (see Figure 2 and Figure 3, and row *HybrMaxMin* in Table 1).

Truth	Class. Negative	Class. Positive
Negative	21277	101251
Positive	3404	1487

The results, reported in Tables 1 and 5, showed that we could deceive the network with high probability, by leading it to classify the modified instances in a way opposite to the true label. The P/R-AUC index on the set made of injected instances indeed drops from 0.42 to 0.06.

Some performance difference between the evolutionary building of pure individuals, where the fitness was evaluated just on the pure individuals, and the classification of the test set made of transformed instances, where each modified function was evaluated, was expected. We avoided it in the experiments we conducted with hybrid individuals. We assessed the performance of the neural network on the test set where, this time, each instance was injected with a code fragment, defined by GE when aimed at changing the classification of a single original given instance.

Such masking of instances, by individuals evolved to perform in the context of a different given function, has been successful since the neural classifier showed a worsened performance measured by a P/R-AUC down to 0.02, as reported in Tables 1 and 6. As a further evidence of the statistical significance of our experiments, we reported, in the last column of the table, the p-values obtained with the Mann-Whitney U test on the different, modified, datasets compared with the original one. In all cases, assuming a significance level $\alpha = 0.01$, the null hypothesis can be rejected, stating that our evolutionary approach is always able to produce solid adversarial examples.

A complete overview of results can be found in Tables from 1 to 6, where more details are reported, in terms of both performance indexes and of confusion matrices for the experiments detailed above (and two additional ones).

Confusion matrices, specifically, highlight how each injection experiment is changing the classification decision for the test instances, towards being wrongly classified as positive or negative, according to our goal for each specific experiment.

5 DISCUSSION

We now discuss our experimental results to show how our technique builds an effective attack to the classification task of a given machine learning system. The specific domain we are considering, namely that of source code classification, does not allow us to find adversarial examples by simply computing a gradient descent (or ascent) considering the output of the network as the loss function, with respect to a continuous input feature space. This is the main challenge, compared to the more common goal of attacking image recognition systems.

Therefore, all the experiments have been designed to overcome this difficulty in exploring the discrete input space of possible source code instances. The results show that this can be done by mean of an evolutionary system, guided by the output of the neural network taken as a fitness evaluation.

To assess the relevance of our results, we can now describe the threat model [26] (or the assessment goals) we are considering, on a given classifying machine learning system:

- the attacker can operate only after the training phase, during testing or after the deployment of the system;
- the attacker knows only a few instances from the test set;
- the attacker can only know the output of the classifier for any input instance he chooses, and this can be considered as a classification probability; the internal structure, parameters and working of the system, instead, are not visible;
- the attacker has the goal of letting the classifier accept, with high probability, vulnerable source code which he does not choose, but which he can alter, so long as the original functionality is conserved.

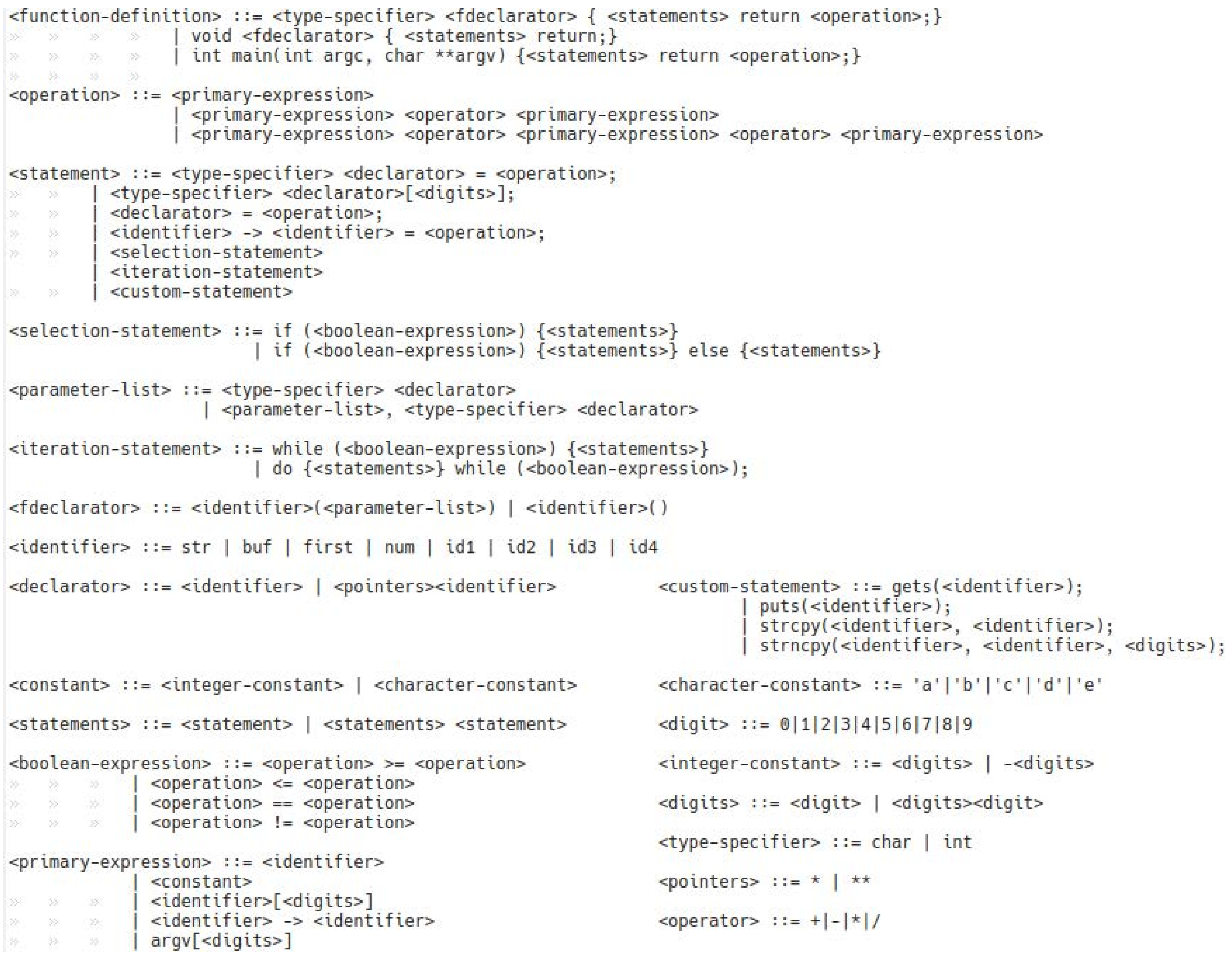
With this setting, our technique allowed us to successfully attack the simple classifier system we chose. Moreover, the low computational effort required to deceive the neural network, and the flexibility of the GE system in successfully adapting to our several experimental settings listed in previous section, suggests that the approach is promising, and that it can be checked against more sophisticated deep learning systems. We expect that for more complex systems, or even non neural classifiers, the exploration of the input space done with our approach could be effective, albeit it could require different parameters that control the evolution of the genetic system.

Moving to the focus of whether the adversarial examples we discovered contributes to the explanation of the neural network behavior, our finding are positive but only preliminary. For instance, we checked whether our minimizing adversarial examples showed preference for including dangerous functions, among those which could be generated by the grammar, such as `strcpy()` or `strncpy()`. We found that such bias, emerging from the training of the network, is apparent, finding that only individuals produced to maximise the classifier output (i.e. making it declare the input instance to be vulnerable) often contains such vulnerable functions, and conversely the opposite is true for minimizing individuals.

6 CONCLUSION AND FUTURE WORK

This work describes a novel approach for exploring the behaviour of a neural network using an evolutionary approach. The proposed application consists in the exploitation of the output of a classifier as the fitness function of a GE algorithm that is able to lead the production of adversarial examples which can deceive the model.

The results obtained are very promising, and thus suggest many further research directions. As a first, immediate extension of this work one can think use the adversarial examples for extending the



```
<function-definition> ::= <type-specifier> <fdeclarator> { <statements> return <operation>;}
»    »    »    »    | void <fdeclarator> { <statements> return;}
»    »    »    »    | int main(int argc, char **argv) {<statements> return <operation>;}
»    »    »    »
<operation> ::= <primary-expression>
                | <primary-expression> <operator> <primary-expression>
                | <primary-expression> <operator> <primary-expression> <operator> <primary-expression>

<statement> ::= <type-specifier> <declarator> = <operation>;
»    »    | <type-specifier> <declarator>[<digits>];
»    »    | <declarator> = <operation>;
»    »    | <identifier> -> <identifier> = <operation>;
»    »    | <selection-statement>
          | <iteration-statement>
»    »    | <custom-statement>

<selection-statement> ::= if (<boolean-expression>) {<statements>}
                        | if (<boolean-expression>) {<statements>} else {<statements>}

<parameter-list> ::= <type-specifier> <declarator>
                   | <parameter-list>, <type-specifier> <declarator>

<iteration-statement> ::= while (<boolean-expression>) {<statements>}
                        | do {<statements>} while (<boolean-expression>);

<fdeclarator> ::= <identifier>(<parameter-list>) | <identifier>()

<identifier> ::= str | buf | first | num | id1 | id2 | id3 | id4

<declarator> ::= <identifier> | <pointers><identifier>          <custom-statement> ::= gets(<identifier>);
                                                                          | puts(<identifier>);
                                                                          | strcpy(<identifier>, <identifier>);
                                                                          | strncpy(<identifier>, <identifier>, <digits>);

<constant> ::= <integer-constant> | <character-constant>        <character-constant> ::= 'a'|'b'|'c'|'d'|'e'

<statements> ::= <statement> | <statements> <statement>          <digit> ::= 0|1|2|3|4|5|6|7|8|9

<boolean-expression> ::= <operation> >= <operation>              <integer-constant> ::= <digits> | -<digits>
»    »    »    | <operation> <= <operation>
»    »    »    | <operation> == <operation>                       <digits> ::= <digit> | <digits><digit>
»    »    »    | <operation> != <operation>
                                                                  <type-specifier> ::= char | int
<primary-expression> ::= <identifier>
               | <constant>                                       <pointers> ::= * | **
»    »    »    | <identifier>[<digits>]
»    »    »    | <identifier> -> <identifier>                     <operator> ::= +|-|*|/
»    »    »    | argv[<digits>]
```

Figure 8: BNF grammar used in all the experiments.

training set and increasing the robustness of the whole model, as in the classical adversarial training techniques.

More in general, the insights and the founding ideas provided in this work can be used for analysing the internal behaviour of a neural network. In the literature, many works highlight the interest in studying the behaviour of single internal neurons of a network [4, 18]. In the field of source code processing, a recent work [29] proposes methods for identifying particular neurons which are able to solve specific tasks (e.g. recognising programs with certain properties) or that are important for the network, regardless of a given task. Our GE-based approach could be used for maximising the activation value of such neurons and thus, eventually, for synthesising program satisfying given specifications.

ACKNOWLEDGMENTS

We would like to thank Luca Manzoni (University of Trieste) for pointing us, in an early discussion, to use Grammatical Evolution for addressing our problem, and the anonymous reviewers for the useful suggestions.

REFERENCES

[1] A. Adadi and M. Berrada. 2018. Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI). *IEEE Access* 6 (2018), 52138–52160. https://doi.org/10.1109/ACCESS.2018.2870052
[2] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML*. 2091–2100.
[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 40:1–40:29.
[4] Fahim Dalvi, Nadir Durrani, Hassan Sajjad, Yonatan Belinkov, Anthony Bau, and James R. Glass. 2019. What Is One Grain of Sand in the Desert? Analyzing Individual Neurons in Deep NLP Models. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*. AAAI Press, 6309–6317.
[5] Cristina David and Daniel Kroening. 2017. Program synthesis: challenges and opportunities. *Philosophical Transactions of The Royal Society A Mathematical Physical and Engineering Sciences* 375 (10 2017).
[6] Quhao Weng Dengsheng Lu. 2007. A survey of image classification methods and techniques for improving classification performance. *International Journal of*

Remote Sensing 28, 5 (2007), 823–870.

[7] Yinpeng Dong, Hang Su, Jun Zhu, and Fan Bao. 2017. Towards interpretable deep neural networks by leveraging adversarial examples. *arXiv preprint arXiv:1708.05493* (2017).

[8] Benjamin S. Duran and Patrick L. Odell. 2013. *Cluster analysis: a survey*. Vol. 100. Springer Science & Business Media.

[9] Y. Fang, S. Han, C. Huang, and R. Wu. 2019. TAP: A static analysis model for PHP vulnerabilities based on token and deep learning technology. *PLoS ONE* 14(11) (2019).

[10] Michael Fenton, James McDermott, David Fagan, Stefan Forstenlechner, Erik Hemberg, and Michael O'Neill. 2017. PonyGE2: grammatical evolution in Python. In *Companion Material Proceedings of Genetic and Evolutionary Computation Conference*. ACM, 1194–1201.

[11] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *International Conference on Learning Representations*. http://arxiv.org/abs/1412.6572

[12] Erik Hemberg, Jonathan Kelly, and Una-May O'Reilly. 2019. On domain knowledge and novelty to improve program synthesis performance with grammatical evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO*, Anne Auger and Thomas Stützle (Eds.). ACM, 1039–1046.

[13] Nwokedi Idika and Aditya P. Mathur. 2007. A survey of malware detection techniques. *Purdue University* 48 (2007), 2007–2.

[14] Asif Karim, Sami Azam, Bharanidharan Shanmugam, Krishnan Kannoorpatti, and Mamoun Alazab. 2019. A Comprehensive Survey for Intelligent Spam Email Detection. *IEEE Access* 7 (2019), 168261–168295.

[15] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP*. ACL, 1746–1751.

[16] John R. Koza. 1992. *Genetic programming: on the programming of computers by means of natural selection*. Vol. 1. MIT press.

[17] Pat Langley. 2011. The changing science of machine learning. *Mach. Learn.* 82, 3 (2011), 275–279.

[18] Quoc V. Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Greg Corrado, Kai Chen, Jeffrey Dean, and Andrew Y. Ng. 2012. Building high-level features using large scale unsupervised learning. In *Proceedings of the 29th International Conference on Machine Learning, ICML*. PMLR.

[19] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen. 2021. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* (2021), 1–1.

[20] Bingchang Liu, Liang Shi, Zhuhua Cai, and Min Li. 2012. Software vulnerability discovery techniques: A survey. In *2012 fourth international conference on multimedia information networking and security*. IEEE, 152–156.

[21] Bing Liu and Lei Zhang. 2012. A Survey of Opinion Mining and Sentiment Analysis. In *Mining Text Data*. Springer, 415–463.

[22] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. 1287–1293.

[23] Anh Nguyen, Jason Yosinski, and Jeff Clune. 2015. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 427–436.

[24] Michael O'Neill, Miguel Nicolau, and Alexandros Agapitos. 2014. Experiments in program synthesis with grammatical evolution: A focus on Integer Sorting. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC*. IEEE, 1504–1511.

[25] Michael O'Neill and Conor Ryan. 2001. Grammatical evolution. *IEEE Trans. Evol. Comput.* 5, 4 (2001), 349–358.

[26] N. Papernot, P. McDaniel, A. Sinha, and M. P. Wellman. 2018. SoK: Security and Privacy in Machine Learning. In *2018 IEEE European Symposium on Security and Privacy (EuroS P)*. 399–414. https://doi.org/10.1109/EuroSP.2018.00035

[27] Erwin Quiring, Alwin Maier, and Konrad Rieck. 2019. Misleading Authorship Attribution of Source Code using Adversarial Learning. In *28th USENIX Security Symposium*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 479–496.

[28] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In *Proceedings of 17th IEEE International Conference on Machine Learning and Applications, ICMLA*. IEEE, 757–762.

[29] Martina Saletta and Claudio Ferretti. 2021. Mining Program Properties From Neural Networks Trained on Source Code Embeddings. arXiv:2103.05442 [cs.SE]

[30] Dominik Sobania and Franz Rothlauf. 2020. Challenges of Program Synthesis with Grammatical Evolution. In *Proceedings of Genetic Programming - 23rd European Conference (EuroGP), held as Part of EvoStar (Lecture Notes in Computer Science, Vol. 12101)*. Springer, 211–227.

[31] Ning Xie, Gabrielle Ras, Marcel van Gerven, and Derek Doran. 2020. Explainable deep learning: A field guide for the uninitiated. *arXiv preprint arXiv:2004.14545* (2020).

[32] Fabian Yamaguchi, Felix "FX" Lindner, and Konrad Rieck. 2011. Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities Using Machine Learning. In *Proceedings of 5th USENIX Workshop on Offensive Technologies WOOT 2011*. 118–127.

[33] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 162:1–162:30.

[34] Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. 2020. Generating Adversarial Examples for Holding Robustness of Source Code Processing Models. In *The 34th AAAI Conference on Artificial Intelligence, AAAI 2020, The 32nd Innovative Applications of Artificial Intelligence Conference, IAAI, The 10th AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI*. AAAI Press, 1169–1176.