

A Deep Learning Approach to Program Similarity

Niccolò Marastoni, Roberto Giacobazzi, Mila Dalla Preda
University of Verona, Italy
{niccolo.marastoni,roberto.giacobazzi,mila.dallapreda}@univr.it

ABSTRACT

In this work we tackle the problem of binary code similarity by using deep learning applied to binary code visualization techniques. Our idea is to represent binaries as images and then to investigate whether it is possible to recognize similar binaries by applying deep learning algorithms for image classification. In particular, we apply the proposed deep learning framework to a dataset of binary code variants obtained through code obfuscation. These binary variants exhibit similar behaviours while being syntactically different. Our results show that the problem of binary code recognition is strictly separated from simple image recognition problems. Moreover, the analysis of the results of the experiments conducted in this work lead us to the identification of interesting research challenges. For example, in order to use image recognition approaches to recognize similar binary code samples it is important to further investigate how to build a suitable mapping from executables to images.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Security and privacy** → *Software reverse engineering*; • **Human-centered computing** → Empirical studies in visualization;

KEYWORDS

Code similarity, deep-learning, obfuscation, code visualization

ACM Reference Format:

Niccolò Marastoni, Roberto Giacobazzi, Mila Dalla Preda. 2018. A Deep Learning Approach to Program Similarity. In *Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis (MASES '18)*, September 3, 2018, Montpellier, France. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3243127.3243131>

1 INTRODUCTION

Binary code similarity is of interest in many different fields within computer science such as malware analysis [18, 21], authorship analysis [1, 6], code patching [7, 11, 13] and copyright infringement investigation [33]. Indeed, with open source projects being available to everyone, the developers take the risk of having their algorithms plagiarized or wrongfully used [33]. The offending parties typically make significant changes to the original code by using automated code obfuscation tools, and then they compile and re-distribute the modified code as binary executables. Thus, in order to detect

plagiarism we need to analyze the similarity stemming from the binaries and not from the source code. When a vulnerability is found in a program deployed all over a company it is important to search for all the occurrences of such vulnerability across all the software installed in the company. The installed software is typically available in binary form and binaries are often obtained through different compiler versions, with different optimization levels, or targeting different architectures. This leads to syntactically different binaries and makes the automatic identification of binary vulnerabilities a challenging problem [11]. Binary code similarity is important also when dealing with malware, as the executable file is usually the only vector of attack. There has been a lot of work in trying to classify malware families using the binaries on different platforms such as Android (e.g., [26, 30]) and Win32 (e.g., [2, 29]). In recent years, researchers have proposed many tools for binary code similarity in different areas of computer science. These tools typically rely on code features extracted either by static or dynamic analysis and on distance metrics defined over these features. These tools naturally inherit the limits of the two analysis approaches: cost of feature extraction and over-approximation of the extracted features for static analysis and cost of feature extraction and code coverage for dynamic analysis. Indeed, an efficient solution to the problem of behavioral binary code similarity analysis is still needed [8, 12]. We postulate that it is necessary to find a measure of program similarity that (1) considers the executable files, (2) recognizes executables with similar behaviour even when their code looks syntactically different (3) does not depend on a priori knowledge of the techniques (obfuscations, compilers, optimizations) used to obtain the diversified executables.

Instead of proposing a new set of code features and distance metrics to measure the similarity between programs, our idea is to investigate whether a sufficiently complex deep neural network can learn this measure of similarity from a wide dataset of labeled programs.

Deep learning has been successfully used in the past to solve problems previously thought impossible to solve by a computer. In 2016 Google unveiled AlphaGo [31], a system based on neural networks that managed to beat some of the top players of Go. The game Go presents some unique challenges, as its gigantic search space has always been a problem for artificial intelligence, but the neural networks managed to learn a strategy that achieved 57% of wins (more than any previous effort) by evaluating more than thirty million moves from human experts. The huge quantity of moves already present in the dataset was not enough to beat the top players, so the neural networks started to learn by continuously playing matches against each other. Another well known example of successful application of neural networks is in image recognition. For example, the ImageNet dataset received its best results by a neural network in 2012 [20], with an experiment that boosted the use of neural networks and general deep learning in the image

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MASES '18, September 3, 2018, Montpellier, France

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-5972-6/18/09...\$15.00

<https://doi.org/10.1145/3243127.3243131>

processing field. The resulting neural network was so complex that the already large dataset of images had to be expanded by a factor of 2048 in order to allow the system to learn and to avoid overfitting.

The underlying power of deep neural networks is shown when there are great quantities of training data, as with a complex enough architecture they have the ability to approximate virtually any kind of function [15]. Most approaches that use a deep learning framework either generate their data [31] or use selected techniques of data augmentation in order to amplify their datasets [20, 32].

In this paper we propose a way to procedurally create arbitrarily-large datasets of behaviourally similar (equivalent) programs, in order to have a dataset that is big enough to train a complex neural network. Our system leverages the Tigress C obfuscator [9] to iteratively apply many layers of syntactic transformations to our original dataset of 47 programs. By definition these obfuscating transformations maintain the semantics of programs while creating hundreds of syntactic variants.

Since there already exists a substantial corpora of works on image processing with deep neural networks, we first investigate whether it is possible to train a neural network to recognize patterns inside the image representation of binaries. To this end we propose a representation that is aligned with other works in the subject [17, 26] and is completely information-agnostic, meaning that we do not use any prior knowledge to enhance the images with features nor do we lose any information during the transformation. We also do not make any assumption about the types of obfuscations used.

The main contributions of this work are:

- A framework based on neural networks capable of classifying code into semantic-equivalence classes by taking as input raw binary images
- Validation of the proposed framework against an automatically generated dataset of binaries stemming from programs obfuscated with the C obfuscator Tigress
- Analysis of the potentials and limitations of this approach and identification of new research challenges

Structure of the paper: Section 2 explores the basics of some of the concepts used throughout this paper, while in Section 3 we start by showing the workflow of the proposed neural network framework for binary similarity and then we provide a description of the elements composing the workflow. Next, in Section 4 we report the results of our experiments and in Section 5 we discuss them. The paper ends with an overview of the next research tasks that we intend to pursue, highlighting the open challenges and research directions opened by our results.

2 BACKGROUND

In this section we briefly introduce some of the key concepts that form the foundation of this paper.

Obfuscation. Let $Prog$ denote the set of all programs. An obfuscation is a program transformation $O : Prog \rightarrow Prog$ that given a program $P \in Prog$ produces a new program $O(P)$ with the same functionality as P but that is “unintelligible” in some sense [3]. Obfuscation techniques are mostly used to deter reverse engineering efforts.

It is always possible to apply a syntactical transformation to obfuscate the code while maintaining its semantic properties. But

for practical reasons, when obfuscating a program we do not want to slow it down too much. Indeed, most obfuscating techniques add computational overhead to the program’s execution, and it just gets worse when multiple obfuscation techniques are used together. A good obfuscation $O(P)$ of a program P should not have more than a polynomial slowdown [3].

This means that, while a program can maintain its original semantics even after being modified ad infinitum by applying more and more obfuscating transformations, it can also get progressively slower, rendering the obfuscation process not entirely palatable. Thus, all semantically equivalent variants of a program can be technically computed, but in a real world scenario not all of them will be, as they would not be usable.

Binary Visualization. Binary files are the raw representation of any type of file stored in the file system. As such they can be numerically represented by any vector that holds these binary values, starting from a simple vector of zeros and ones to other matrices representations that groups these values in other meaningful ways. These matrices can then be translated into images by encoding every pixel with the value held in the respective cell of the matrix. If the matrix is purely binary, the resulting image will only hold two colors (usually black and white).

Classification. In machine learning a typical classification problem consists of a dataset of samples that can be divided in different classes by considering certain metrics. For example, a set of images can be grouped with regards to their main colors by using some simple quantitative measure. Or they could be grouped according to the subject portrayed in the image, and this is a much harder problem to solve [20].

In our case the dataset is made up of different programs that can be grouped with respect to their semantics, meaning that we group together programs that compute the same function. This means that our classifier ideally has to label the code samples with regard to their input-output relationships, disregarding their syntactic dissimilarities. Or, more realistically, it should find syntactic patterns that survived the many iterations of obfuscation, since a true semantic similarity would be undecidable (Rice’s Theorem).

Building such a classifier would require us to define a representation of the code and a similarity measure that ensures the correct grouping of semantically similar programs.

Deep Learning. DL groups together multiple machine learning techniques that are characterized by multiple layers of connected non-linear units. Their use has revolutionized the computer vision field [23], where in some cases they have achieved better accuracies than human experts.

In our case we will work mostly with a Convolutional Neural Network (CNN), that is characterized by multiple convolutional layers each usually followed by a pooling layer that provides down-sampling and a ReLU layer for our non-linear activation function. The convolutional architecture is especially suited for the classification of unprocessed images, allowing the system to learn the correct problem representation by itself [22]. We use the shortening NN when we mean a general Neural Network, as opposed to our CNN.

3 EXPERIMENTAL SETUP

In Fig. 1 we depict the workflow of the deep learning approach that we propose for binary similarity analysis through the visualization of binaries.

We start by considering an initial dataset of 47 simple programs written in C. This initial dataset is then iteratively obfuscated with the C obfuscator Tigress, creating hundreds of obfuscated C programs that we can group in 47 semantic equivalence classes with regard to the program from which they are generated. These obfuscated programs are compiled to obtain a dataset of obfuscated binaries classified in 47 semantic equivalence classes. Every executable file in the obfuscated dataset is transformed into an image (see Fig. 2) and is then fed into a convolutional neural network (CNN). The CNN is then trained to classify these samples into their respective semantic equivalence classes.

In the following we describe the elements of our framework in more detail.

3.1 Initial C Dataset

In order to gather a dataset of semantically distinct programs to train our NN, we downloaded two sets of programs written in C: one set consists of 32 simple C programs used to learn the C programming language and the other set contains 15 C programs downloaded from the Google Code Jam competition. We also consider the popular MNIST dataset to ensure that our approach works with standard image recognition problems. The MNIST dataset [24] contains 60 000 samples of hand-written digits, already encoded in simple 28×28 matrices with float values between 0 and 1 representing grey-scale pixels. This dataset has been extensively used in the pattern recognition community ever since its inception, and it can be used as a cheap benchmark since it does not require any pre-processing and it has an established ground truth.

In the following we report the list of the 32 simple C programs downloaded from www.programiz.com/c-programming/examples/:

- | | |
|------------------------|-----------------------------|
| (1) armstrong_n.c | (17) n_is_prime.c |
| (2) calculator.c | (18) n_is_sum_of_primes.c |
| (3) char_frequency.c | (19) positive_or_negative.c |
| (4) count_digits.c | (20) power_n.c |
| (5) count_vowels.c | (21) prime_n_intervals.c |
| (6) factorial.c | (22) pyramid.c |
| (7) factorial_rec.c | (23) quotient_remainder.c |
| (8) factors.c | (24) remove_char.c |
| (9) fib_1.c | (25) reverse_integer.c |
| (10) fib_2.c | (26) store_struct.c |
| (11) gcd.c | (27) strcat.c |
| (12) gcd_rec.c | (28) strcpy.c |
| (13) hello_world.c | (29) stringsort.c |
| (14) lcm.c | (30) strlen.c |
| (15) leap_year.c | (31) sum.c |
| (16) n_is_palindrome.c | (32) times_table.c |

These programs perform semantically distinct duties and they are small, averaging only 23 lines of code. This leads to small binaries after compilation, which is an important factor since it allows us to feed the entire binary image to our NN without going over

our strict memory boundaries (6GB of dedicated RAM in our GPU) and thus without losing information.

Google Code Jam is an international programming competition where contestants are given a problem to solve in their language of choice. To check whether our approach works with bigger programs and to add more complexity to the problem at hand, we downloaded some solutions in C from this competition. Here is the list of the 15 samples downloaded from Google Code Jam contests between 2008 and 2011, all of which contain 78 lines of code on average :

- | | |
|------------------------------|----------------------------|
| (1) alien_language.c | (9) rotate.c |
| (2) bot_trust.c | (10) saving_the_universe.c |
| (3) candy_splitting.c | (11) snapper_chain.c |
| (4) fair_warning.c | (12) theme_park.c |
| (5) fly_swatter.c | (13) train_time_table.c |
| (6) magicka.c | (14) watersheds.c |
| (7) minimum_scalar_product.c | (15) welcome_to_codejam.c |
| (8) multibase_happiness.c | |

Thus, the resulting initial C dataset is composed by the above listed 47 C programs that perform a wide range of operations. These programs form the 47 labels of our classification problem.

3.2 Tigress

The Tigress C Obfuscator is a tool that operates at the C source code level, leveraging the CIL [27] system for the transformations. The tool offers several syntactic transformations that can be stacked together in order to improve the efficiency of the obfuscation. In our experiments we consider the following eight obfuscations of Tigress:

- **Flatten** implements code flattening by completely removing the original control flow structure of the program and replacing it with a switch statement [35]. The switch control variable is then modified dynamically to decide the order in which the basic blocks are executed, which produces a "flattened" control flow.
- **Split** performs a split of a specified function in different functions that when combined perform the same task as the original.
- **EncodeArithmetic** Integer arithmetic is replaced with more complex expressions. For example the expression $z = x + y + w$ can be replaced by $z = (((x^y) + ((x \wedge y) \ll 1)) \vee w) + ((x^y) + ((x \wedge y) \ll 1)) \wedge w$. There are many of these transformation (all taken from the book Hacker's Delight [36]) and they are chosen randomly at each run.
- **InitOpaque** adds data structures with invariants used to add opaque predicates.
- **EncodeLiterals** replaces literal strings with functions that generate them at runtime.
- **InitEntropy** creates the variables needed to add randomness during execution.
- **InitImplicitFlow** initializes the signal handlers for other transformations that rely on implicit flow.
- **RandomFuns** adds random functions to the original code.

We denote with \mathbb{T} the set of the eight obfuscations of Tigress described above.

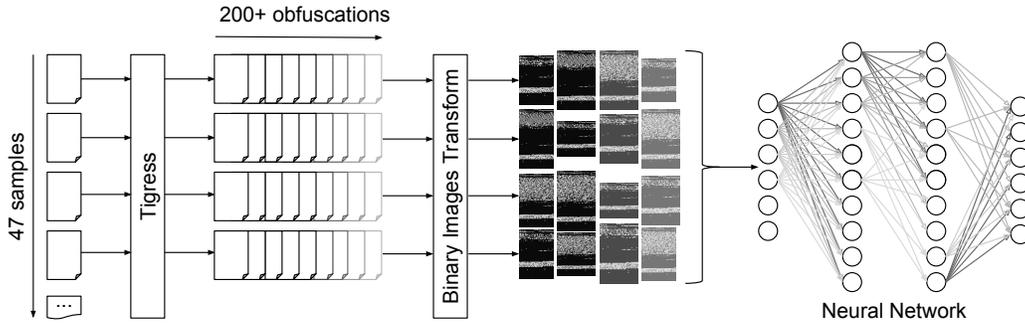


Figure 1: Workflow of the proposed binary similarity analysis framework.

3.3 Obfuscated Binaries Dataset

Deep learning has shown its potential when dealing with unprocessed data [23], and produced the best results when dealing with huge multi-labeled datasets of images [20]. We leverage this potential by automatically generating very big datasets of binary images and feeding them to our NNs. In particular, we use Tigress to generate the obfuscated binaries dataset from the initial C dataset. We start by obfuscating the 47 C programs of the initial dataset with obfuscations from the set \mathbb{T} , thus creating 8 variants of each original C program. Then the obfuscations are stacked with two layers to create an additional 56 variants and so on.

In order to create more meaningful stacks of obfuscations we do not allow two identical transformations to be stacked on top of each other (a Flatten followed by another Flatten would not be very interesting), so we use $\binom{8}{k}$ combinations without repetitions with k that ranges between 0 and 8. Moreover, the order in which obfuscations are applied is important, since different orders can modify the binary in different ways (e.g. a Flatten followed by a Split would look very different the other way around). For this reason we consider every permutation resulting from the combination. We denote with $Perm(\mathbb{T}, k)$ ranged over by π , the set of possible permutations of obfuscations from \mathbb{T} of length at most k with no repetitions. Here are some examples of the allowed permutations:

- π_1 : InitOpaque, Flatten
- π_2 : Split, EncodeLiterals, EncodeArithmetic
- π_3 : Split, EncodeArithmetic, InitOpaque
- π_4 : EncodeLiterals, Split, EncodeArithmetic
- ...

We define the \mathbb{T} -equivalence relation on programs as $\equiv_{\mathbb{T}} \subseteq Prog \times Prog$ where given two programs $P_1, P_2 \in Prog$ we have that $P_1 \equiv_{\mathbb{T}} P_2$ if and only if there exists two permutations $\pi_1, \pi_2 \in Perm(\mathbb{T}, k)$ and a program $P \in Progr$ such that $P_1 = \pi_1(P)$ and $P_2 = \pi_2(P)$. We denote with $[P]_{\mathbb{T}}$ the \mathbb{T} -equivalence class of a program $P \in Prog$, namely the set of all programs equivalent to P with respect to the obfuscations allowed by $Perm(\mathbb{T}, k)$. When considering the set of obfuscated C programs obtained by applying obfuscations in $Perm(\mathbb{T}, k)$ to the C programs in the initial dataset we have a \mathbb{T} -equivalence class for each C program in the initial dataset. These equivalence classes contain programs that exhibit the same behavior while being syntactically dissimilar, providing us with a trusted

ground truth. After these classes have been created, they serve as labels to train the NNs for our classification problem.

Since applying obfuscations takes a lot of time, we generate only 200 variants for every initial program, in this way we obtain 9 400 obfuscated C programs grouped into 47 \mathbb{T} -equivalence classes. The dataset of binaries obtained by compiling these C programs is called `FinalObfuscatedDataset`, while the smaller dataset containing 6 400 obfuscated binaries coming from the initial 32 simple C programs is called `SimpleObfuscatedDataset`.

The `FinalObfuscatedDataset` was created in roughly 45 minutes on a Ubuntu system with a new generation i7 and 32GB of RAM.

3.4 Binary Visualization

Each executable in the obfuscated binaries dataset is imported as a raw file within our Python tool and it is converted using the `numpy` package in order to represent it as a list of hexadecimal numbers (16^4 max value). It is then possible to convert these lists of hexadecimal numbers to an image by assigning a width of 64 (we discuss this seemingly arbitrary number in the next section) and using the hexadecimal numbers as a value for the pixel's color. For example, Fig. 2 was generated by the `imshow()` function in Matplotlib using the `jet` cmap.

In the following we discuss the issues that arise when representing binaries as images if we want to feed them to a NN.

Set Width. Binary files are mostly contiguous sequences of bytes and they do not have much of a structure. For this reason when representing them as images they may vary in width and height. However, in order to perform image recognition we need data to be visually consistent: it needs a fixed width and a fixed height. In [17] the authors decided to set the width of every image with a fixed value of 128, while Nataraj et al. [26] chose to vary the width of the image according to its size, but still selecting it within a set of powers of 2. We experimented with many settings and found that they greatly affect the results, we discuss this in section 4.2.

Data Normalization. Binary files have random sizes, which means that when a fixed width w is applied we also need to ensure that the size of the file is divisible by the width. So the first operation that is applied to the data is *padding*: we add zeros to the end of every file until we reach a number that is divisible by w .

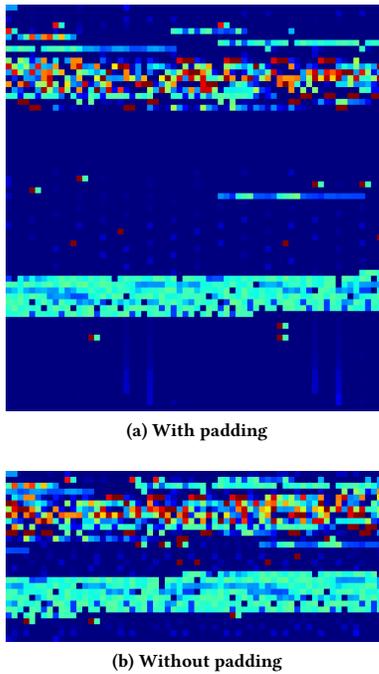


Figure 2: Normal armstrong_n and its version without zeros

Having the same width is not enough for an image recognition problem, every image in the dataset is usually required to have the same size. There are many approaches to this task, usually using some form of random crop selection or resizing and deforming the images.

In our case we implemented three simple forms of normalization:

- (1) **lower** The length of the smallest sample (min_length) is used as length for every other sample whose length is cut at min_length (see Fig. 3). This of course results in faster training times (up to 10x faster) but at the same time causes loss of information and affects the results, in some of the cases by quite a margin.
- (2) **upper** The length of the biggest sample (max_length) is used as length for every other sample. To achieve this we add 0s to the end of the binary (see Fig. 3). This is a transformation that does not lose any information and thus grants the best accuracy in every case, at the cost of worse performance.
- (3) **mean** The mean normalization is used when upper cannot be used while analyzing the final dataset, because of memory constraints on both the system RAM (32 GB) and the GPU RAM (6 GB). The length of the images is set to the smallest value between $(max_length + min_length)/2$ and 596. This value is the biggest allowed length, since any higher value will effectively crash the CNN, exceeding the GPU's 6GB RAM limit.

In Section 4.2 we explore how the different normalizations affect the considered classification problem.

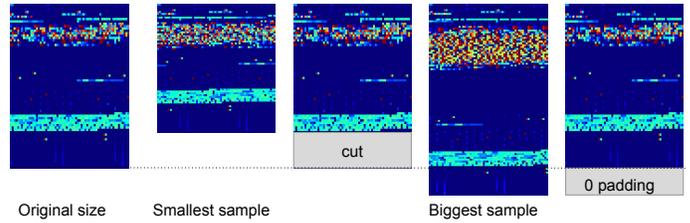


Figure 3: Normalizations

3.5 The Neural Network

The implementation of our CNN is entirely coded in Python using the popular library `TensorFlow`. The CNN takes as input the matrices representing the pixels in the binary image, and it is built to allow for maximum flexibility, since the size of the images in a dataset is always fixed, but different normalizations will change the height of the inputs.

The CNN has then been trained on a desktop computer running Ubuntu 16.04 with a last generation i7, an nVidia Geforce 1060 with 6GB of dedicated memory and 32 GB of RAM. Using `TensorFlow` allowed us to leverage the full power of the GPU in our system during the training of the Neural Net, since it creates the computational graphs before deferring the calculations to the video card.

The architecture of our CNN is standard, it has an input layer, followed by two convolutional layers, each followed by a maxpooling layer. At the end we added a fully-connected layer just before the output. The input layer has a dynamic size, meaning that it depends on how many samples are in the dataset (x axis) and how big these samples are (y axis). Each experiment has different values because of the nature of our problem, since we work with different datasets and some obfuscations will add length to the resulting binary.

The first convolutional layer applies a filter of size 5×5 with a stride of 1, using a rectified linear unit as activation function. The output of the convolution is then applied as input to a simple maxpooling layer that will downsample the data by focusing on tiles of pixels of size 2×2 . After this layer we apply a second convolution layer followed by the last maxpooling layer.

All the weights in the CNN are initialized with random values taken from a normal distribution with standard deviation set at 0.1, to avoid running into local minima at early stages. The biases have a fixed value of 0.1 in order to let the ReLU functions fire in the beginning.

Several parameters can be tuned at every algorithm run, such as:

- (1) **test_ratio** This value is usually set to 0.1, which means that 10% of the dataset (selected randomly) is going to be reserved for the test set, while the remaining 90% will train the NNs. We experimented with this parameter and tuning it to higher values doesn't significantly change the accuracy in the test set until a 0.4 threshold, where it gives slightly worse results.
- (2) **gradient_rate** Tuning this parameter allows the gradient descent optimizer to perform longer or shorter strides. Since the algorithm can get stuck in local minima we found that a rate of 0.8 gave us the best results, while the beginning value

of 0.5 often limited the training process to low accuracies and was highly reliant on the initial random seed.

- (3) **limit** This is the number of iterations during the NN training, useful to set an early stop in case of overfitting and to check whether longer training times could improve the accuracy. Our best results have been with values higher than 4 000, so it is usually set to 10 000.
- (4) **random_seed** In order to better randomize the split between the test and the train set we allow the random seed to be changed at every run.
- (5) **norm** This parameter represents the normalizations and can be chosen between *lower*, *upper* and *mean*. We will expand on the significance of these values in Section 4.2.

The training is guided by the cross entropy function, that takes into account the predicted labels and the ground truth, while the gradient descent optimizer (built into TensorFlow) decides the direction in which the coefficients have to be modified in order to reduce the mean error.

Several metrics are coded within the learning tool, at the end of every run we save a plot of the train and test accuracy and the evolution of the confusion matrix. This allows us to better understand where the problems lie in the learning process.

3.6 Evaluation

The datasets are randomly divided into training set and test set at the beginning of the algorithm run with a set ratio of 90% training and 10% test. In order to ensure a correct cross-validation we run the algorithm multiple times with different random seeds (thus changing the test set at each run) and the results are evaluated with regard to the accuracy value averaged over multiple runs. The NN is then trained exclusively with the samples from the training set, while the test set samples are always used to check whether the learning algorithm generalizes. It is important to never train the NN with samples from the test set to ensure that the classification generalizes and to detect overfitting. The *accuracy* is defined as the number of correct predictions over the number of samples. The accuracy is measured both in the train set and in the test set, but in the train set it is merely used as a mean to check whether the NN is actually able to learn from the dataset. Every accuracy value that is reported in Section 4 is relative to the test set, since it is the only measure that allows us to ensure the correct generalization of the resulting classifier.

In the evaluation process we also use *confusion matrices* as a graphical tool to check whether some classes are being confused with other classes and intuitively understand which ones are easier to recognize. We prefer this visualization instead of precision and recall since these measures would need to be averaged over 47 classes, hiding useful information about each class. In Fig. 4 we show the confusion matrix for the final dataset when using the *mean* normalization and then one with the *lower* normalization. The x axis represents the predicted classes and the y axis holds the actual classes, then every cell of the matrix is assigned a darker color in relation to the number of samples that it represents.

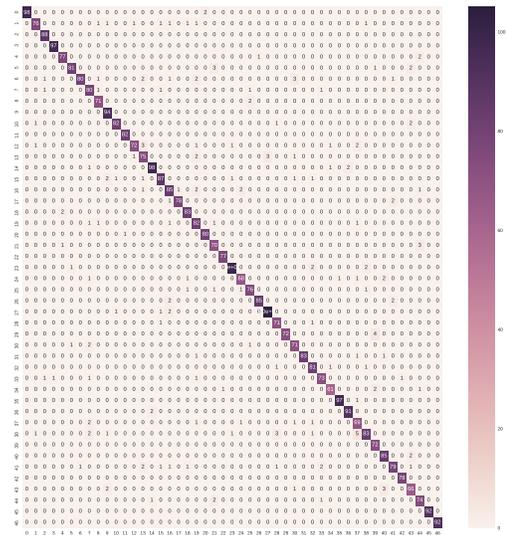


Figure 4: Confusion Matrix for the FinalObfuscatedDataset, a successful classification with the 'mean' normalization

4 RESULTS

In this section we report the results of our experiments, highlighting how the various parameters affect the classification.

4.1 Width

Our choice of setting the width to 64 is made mainly because it is a power of 2 and represents the maximum bit size that an instruction can take in a 64 bit system such as the one we work on. This ultimately proved to be a good choice, since this parameter greatly affects the learning process.

In order to investigate whether the width of the images affect the classification accuracy in a simple image recognition problem, we held some experiments with the MNIST dataset in our system. The differences in the prediction scores are not significant and the CNN is able to recognize the digits even if they are scrambled in a way that no human could recognize them anymore (see Table 1 and Fig. 5).

We perform the same tests also on our dataset of images extracted from obfuscated binaries. The results reveal a negligible difference in both train and test accuracy only if the width of the binary image is changed from 64 to 32. While when considering other width values (still powers of 2) the classification accuracy drops, and it reaches 0.04 in the test set with a set width of 256, an accuracy that is slightly better than random guessing (see Table 2). This is one of the first observations that leads us to believe that our classification problem is vastly more complex than simple digits recognition.

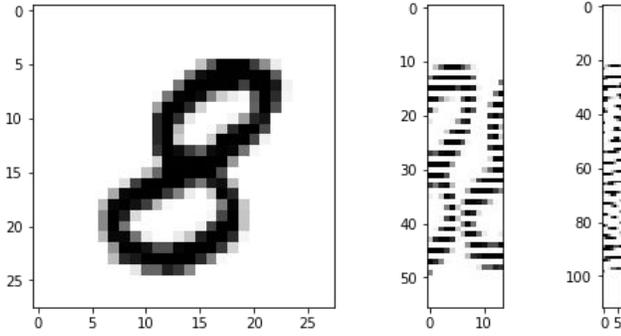


Figure 5: A digit from the MNIST dataset with varying widths: 28x28, 56x14 and 112x7

Table 1: Width change in MNIST

MNIST	28x28	56x14	112x7	14x56	7x112
train	0.98	0.98	0.96	0.98	0.96
test	0.98	0.97	0.96	0.97	0.96

Table 2: Width change in SimpleObfuscatedDataset

OBF	?x32	?x64	?x128	?x256
train	0.98	0.97	0.43	0.10
test	0.89	0.91	0.33	0.04

4.2 Shallow Neural Net

We built a simple NN without any convolutional layers in order to test the smallest size needed to achieve good results in a simple image processing task. We only have the input layer, with the biases and the weights initialized exactly like the deeper net in our study. The 1-layer NN was trained with the MNIST dataset and achieved an accuracy of 0.92, verifying that it is well suited for a simple image recognition task. We note that 0.92 is by no means a good result when it comes to this particular problem, since the state of the art classifiers can get up to 0.99 accuracy. Our focus is not to improve the image recognition state of the art but merely to show that the same techniques can be applied in our specific problem setting.

We observe that binary files are full of zero-padding, which does not add any relevant information about the program run but affects the size of the extracted image. For this reason in our experiments we also consider the case where these zeros are removed. So when presenting our results we have a parameter called **Zeros** that we set to *T* when we consider binaries with zero-padding and to *F* when we consider binaries without zeros. Removing zeros does not add nor detract information and this is confirmed by our experiments on the SimpleObfuscatedDataset where the results with or without zeros are the same, with statistically irrelevant variations (see Table 4).

After being trained with the SimpleObfuscatedDataset, the shallow NN predictions had an accuracy of 0.03, which is just slightly

Table 3: Accuracies on the SimpleObfuscatedDataset with the Shallow Neural Net

Norm	lower		mean		upper	
	T	F	T	F	T	F
Zeros						
Accuracy	0.02	0.03	0.03	0.03	0.02	0.03

Table 4: Accuracies on the SimpleObfuscatedDataset with our Convolutional Neural Network

Norm	lower		mean		upper	
	T	F	T	F	T	F
Zeros						
Accuracy	0.86	0.84	0.91	0.91	0.94	0.93

better than randomly guessing the program labels (see the left side of Table 3).

Fig. 6 shows the test accuracy in an algorithm run with 10 000 iterations, where the system clearly improves the test accuracy over time with the MNIST dataset but fails to move from the baseline with the SimpleObfuscatedDataset. The 0.03 value was encountered both in the train set and in the test set, meaning that the algorithm not only fails to generalize, but is unable to train at all. A failure to improve the train accuracy usually means that the NN is not expressive enough for the problem at hand.

This led us to believe that our binary images recognition problem might be of a higher level than simple digits recognition.

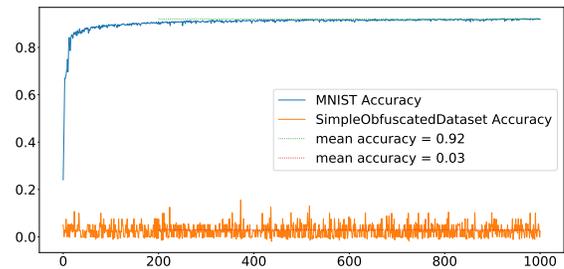


Figure 6: Plot of the test accuracy with the Shallow Neural Network on the MNIST dataset (blue) and with our SimpleObfuscatedDataset (orange)

4.3 Deep Convolutional Neural Net

Our second NN has a design still suited for simple image recognition. To test this we ran the algorithm with the MNIST dataset and scored 0.98 on both the train set and the test set with 10 000 iterations.

The SimpleObfuscatedDataset is then used as input to the CNN and it achieved a test accuracy of 0.94 (see Table 4) with a dataset of 6 400 samples divided into 32 labels and the following settings: (1) normalization: upper (2) zeros: True.

The FinalObfuscatedDataset after 20 000 iterations achieved a test accuracy of 0.88 (see Table 5 and Fig. 7) with a dataset of

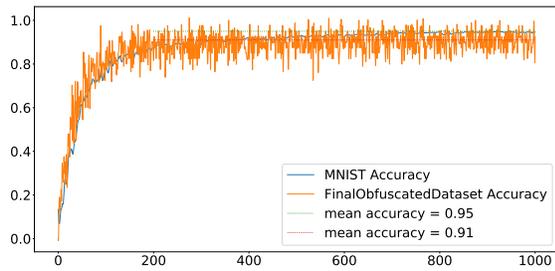


Figure 7: Plot of the test accuracy with the Convolutional Neural Network on the MNIST dataset (blue) and with our FinalObfuscatedDataset (orange)

Table 5: Accuracies on the FinalObfuscatedDataset given as input to our CNN

Norm	lower		mean		upper	
Zeros	T	F	T	F	T	F
Accuracy	0.04	0.03	0.74	0.88	N.A.	N.A.

9400 samples divided into 47 labels and the following settings: (1) normalization: mean (2) zeros: False.

In this case we used the 'mean' normalization, as 'upper' produced images that our system could not handle due to their large size. The overall training of the CNN takes approximately one hour.

This means that our CNN correctly classifies 88% of the samples that are contained in the FinalObfuscatedDataset into their respective equivalence classes.

5 DISCUSSION

In this section we want to discuss what we can learn from the results that we have obtained in our experiments.

5.1 Semantic Similarity

As reported in Table 5, our approach correctly classifies 88% of the samples in the FinalObfuscatedDataset in their respective equivalence classes by considering only the images extracted from their executables. This dataset was created by applying the obfuscations in a uniformly distributed order (considering every permutation), which ensures that there are no biases stemming from artificially denser data regions.

Since our NN can recognize programs whose syntax has been scrambled, it can recognize the semantic similarity of some programs that calculate the same functions.

5.2 A Different Problem Class

As shown in Fig. 6, the shallow NN failed to learn on our dataset and succeeded on MNIST, but it might be argued that recognizing digits in a dataset that has already been preprocessed is possibly the easiest image recognition problem out there. This is one of the reasons why our problem could be closer to harder problems like facial recognition or object recognition.

Furthermore, classifying images of binaries is a problem where humans would not fare well at all, making it distinct from most image recognition tasks.

5.3 Visualization Techniques

Our transformation of the binary file into an image is a lossless one, but due to limited resources we were forced to implement two different types of manipulations: zero removal and height normalization.

In every experiment with both datasets we varied the parameters to see whether the removal of information would impact the classification accuracy. The lower normalization always gives the worst performance (as far as the accuracy goes, but the speed of the training process greatly improves), while the upper normalization gives the best results when available.

When removing the zeros from the code we encountered no significant loss of precision in the classification while working with the small programs in the SimpleObfuscatedDataset (see Table 4), but there is a big difference between the classification of the FinalObfuscatedDataset with zeros and without zeros as can be seen in Table 5. Since the average length of the files in the FinalObfuscatedDataset greatly exceeds our limit of 586, this value is used as fixed height for the images. This means that removing the zeros will in fact allow more actual information to fit into smaller images (as seen in Fig. 2).

Considering both factors, it is clear that our approach works best when the mapping between the binary and its image is as close to lossless as possible.

5.4 Source to Binary Loss of Obfuscation

Compilers employ several optimizations when generating the executable from the source, which means that not all the syntactic transformations on the source code will be still present in the binary. Different compilers also employ different optimization techniques, thus we restricted our study to gcc in order to control for this variation.

5.5 Limitations

Our study focuses on small programs because the CNN needs to look at the entire image extracted from the binary to better classify every executable in its class. Applying multiple obfuscations easily expands the code and the resulting compiled file, forcing us to cut the images in the experiments with the final dataset.

This means that the size of the executable file is the biggest limitation of this study. Some possible solutions include (1) Using external cloud platforms to increase the available memory (2) applying preprocessing techniques to reduce the size of the images.

6 RELATED WORK

In this section we briefly describe closely related work.

In [25], the authors propose the use of the longest common sequence of semantically equivalent basic blocks as a similarity measure, where this semantic equivalence is checked by a theorem prover after the input-output relations of the blocks have been mapped into a set of symbolic formulas. Symbolic execution is also employed in [39] in order to find similarity between binaries. Both

of these approaches suffer from the usual drawbacks of symbolic execution tools and constraint solvers, such as huge computational overheads and difficulty in handling certain specific syntactic features, such as indirect jumps.

The work presented in [11] introduces the idea of *similarity by composition* relative to binary similarity. Their technique draws inspiration from image similarity works [4], where the images are deemed similar if some of their respective regions are similar. Like our approach, their system does not need the source code in order to extract information, even though they do not work on the raw binary itself but on its decompiled code. The goal to detect similarity in binaries from multiple architectures is shared by [16], where the authors propose another semantic approach.

The visualization of executable files has been used to help analysts explore the binaries visually, to spot recurring patterns [17] that can help recognize different types of packers or to identify the main fragment types in different binaries [10]. Nataraj et al. [26] developed a technique to detect malware samples belonging in the same family by processing grey-scale images extracted from executable Win32 files. Our study uses the entire image, while they focus on a vector of 320 features extracted from the images, the *gist* [28], which uses a wavelet decomposition of an image. Another difference is the classification method, while we use Deep Convolutional Neural Networks they use a k-nearest neighbors classifier with a euclidean distance. The dataset and problem at hand is also not related to ours, since they focus on malicious Win32 executables and their only obfuscated samples exhibit a simple section encryption that does not foil the textural features.

None of the previous works assume an agnostic approach nor do they use Neural Networks. In [37] deep learning is used to detect clones by extracting information from the source code, but it is focused on reusability and avoiding the repetition of existing code fragments in a codebase. Another application of deep learning techniques can be found in [19] where the authors propose a novel approach for malware classification using system call sequences. In [34] deep learning is shown to be suitable for software engineering tasks, in particular code clone detection, using different representations. These works do not use features extracted from the binaries.

Other works apply similarity measures to code in order to group different programs by author, where the most promising results used a very large corpora of features extracted from abstract syntax trees in order to de-anonymize up to 1600 authors with 94% accuracy [5]. Lately this approach has been successfully ported to binaries in [6].

7 CONCLUSIONS AND FUTURE WORK

In this paper we propose a novel approach for program similarity detection that uses raw binary files, allows for syntactic differences and does not depend on further a priori knowledge. We build a tool that leverages the power of a deep convolutional neural network in order to classify the images extracted from these binary files. We tested this approach on a dataset consisting of 9 400 programs obfuscated iteratively with the C obfuscator Tigress starting from 47 original ones. The classifier achieved an accuracy of 0.88 on

the test set, showing that the technique is promising for similarity detection.

7.1 Future Work

The positive results of our approach opened some interesting possible lines of research. In this section we briefly list some of the most compelling open challenges.

7.1.1 New Visualization Techniques. We have started investigating what happens when we lose information, both by eliminating zeros and cropping the image in order to focus on the first part of the binary. Although new forms of data normalizations are definitely needed, it would also be interesting to investigate what would happen with different types of transformations. Is it possible to transform a binary into an image while adding information that is learnt with either static or dynamic analysis? Would this make the learning process better?

7.1.2 Formalize Transformations from Binary to Image. We would like to take a formal approach regarding the transformations from an executable to an image, accounting for loss of information or entropy.

7.1.3 Semantic Similarity. Since our NN can recognize programs whose syntax has been modified, we can test whether it can recognize the semantic similarity of programs that calculate the same functions, but written by different coders.

7.1.4 Learning Obfuscations. One of the reasons why we worked with the Google Code Jam datasets was to have a large sample set of C programs so that we could try to learn the shape of the obfuscations as well. This would be useful for reverse engineering. There are existing works [14] that take a packed binary as input and try to establish which packer has been used by detecting some of the obfuscation techniques used.

7.1.5 Learning Authorship. Different programmers writing the same function will usually approach the problem in a different way, thus the same program written by different people is technically a syntactic transformation that maintains the semantics of the original function, not unlike obfuscations. Connected to the point above, if we can learn the obfuscation patterns, can we learn the authorship?

7.1.6 More Obfuscators. Our approach should be completely agnostic to the language used, since we work directly with the compiled binary, but would it work on different types of binaries, like Java? And would it work with different chains of obfuscations? Our next steps will be directed towards testing the limits of what our approach can recognize so far.

7.1.7 What is the CNN Actually Learning? Many techniques employed in image processing with Neural Networks allow the researchers to look more closely into what the system is actually learning by visualizing what the neurons are recognizing [20, 38]. In typical image recognition problems these results show what kind of edges and other kinds of higher level features are learnt, in our case it could show the patterns in the binary code that survive the obfuscations, allowing the classifier to work.

ACKNOWLEDGEMENTS

We would like to thank our reviewers for their valuable comments and input to improve our paper.

REFERENCES

- [1] Saeed Alrabaae, Noman Saleem, Stere Preda, Lingyu Wang, and Mourad Debbabi. 2014. Oba2: An onion approach to binary code authorship attribution. *Digital Investigation* 11 (2014), S94–S103.
- [2] Michael Bailey, Jon Oberheide, Jon Andersen, Z Morley Mao, Farnam Jahanian, and Jose Nazario. 2007. Automated classification and analysis of internet malware. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 178–197.
- [3] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. 2001. On the (im) possibility of obfuscating programs. In *Annual International Cryptology Conference*. Springer, 1–18.
- [4] Oren Boiman and Michal Irani. 2007. Similarity by composition. In *Advances in neural information processing systems*. 177–184.
- [5] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. 2015. De-anonymizing programmers via code stylometry. In *24th USENIX Security Symposium (USENIX Security)*, Washington, DC.
- [6] Aylin Caliskan-Islam, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. 2015. When coding style survives compilation: De-anonymizing programmers from executable binaries. *arXiv preprint arXiv:1512.08546* (2015).
- [7] Mahinthan Chandramohan, Yinxiang Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 678–689.
- [8] Jonathan D Cohen. 2018. Apparatus and method for identifying similarity via dynamic decimation of token sequence N-grams. US Patent 9,910,985.
- [9] Christian Collberg. 2015. The Tigress C diversifier/obfuscator. Retrieved August 14 (2015), 2015.
- [10] Gregory Conti, Sergey Bratus, Anna Shubina, Andrew Lichtenberg, Roy Ragsdale, Robert Perez-Aleman, Benjamin Sangster, and Matthew Supan. 2010. A visual study of primitive binary fragment types. *White Paper, Black Hat USA* (2010).
- [11] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical Similarity of Binaries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 266–280.
- [12] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 392–404.
- [13] Debin Gao, Michael K. Reiter, and Dawn Song. 2008. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *Information and Communications Security*. Springer Berlin Heidelberg, 238–255.
- [14] Nguyen Minh Hai. 2016. A STATISTICAL APPROACH FOR PACKER IDENTIFICATION. *Vietnam Journal of Science and Technology* 54, 3A (2016), 129.
- [15] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. 1989. Multilayer feed-forward networks are universal approximators. *Neural networks* 2, 5 (1989), 359–366.
- [16] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2017. Binary code clone detection across architectures and compiling configurations. In *Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on*. IEEE, 88–98.
- [17] Ashutosh Jain, Hugo Gonzalez, and Natalia Stakhanova. 2015. Enriching reverse engineering through visual exploration of Android binaries. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. ACM, 9.
- [18] Jiyong Jang, David Brumley, and Shobha Venkataraman. 2011. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. ACM, New York, NY, USA, 309–320.
- [19] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. 2016. Deep learning for classification of malware system call sequences. In *Australasian Joint Conference on Artificial Intelligence*. Springer, 137–149.
- [20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.
- [21] Arun Lakhotia, Mila Dalla Preda, and Roberto Giacobazzi. 2013. Fast location of similar code fragments using semantic 'juice'. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop 2013, PPREW@POPL 2013, January 26, 2013, Rome, Italy*. ACM, 5:1–5:6.
- [22] Yann LeCun, Yoshua Bengio, et al. 1995. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks* 3361, 10 (1995), 1995.
- [23] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436.
- [24] Yann LeCun, Corinna Cortes, and CJ Burges. 2010. MNIST handwritten digit database. *AT&T Labs [Online]*. Available: <http://yann.lecun.com/exdb/mnist> 2 (2010).
- [25] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 389–400.
- [26] Lakshmanan Nataraj, S Karthikeyan, Gregoire Jacob, and BS Manjunath. 2011. Malware images: visualization and automatic classification. In *Proceedings of the 8th international symposium on visualization for cyber security*. ACM, 4.
- [27] George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. 2002. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction*. Springer, 213–228.
- [28] Aude Oliva and Antonio Torralba. 2001. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International journal of computer vision* 42, 3 (2001), 145–175.
- [29] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. 2008. Learning and classification of malware behavior. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 108–125.
- [30] A-D Schmidt, Rainer Bye, H-G Schmidt, Jan Clausen, Osman Kiraz, Kamer A Yuksel, Seyit Ahmet Camtepe, and Sahin Albayrak. 2009. Static analysis of executables for collaborative malware detection on android. In *Communications, 2009. ICC'09. IEEE International Conference on*. IEEE, 1–5.
- [31] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484–489.
- [32] Patrice Y Simard, David Steinkraus, John C Platt, et al. 2003. Best practices for convolutional neural networks applied to visual document analysis. In *ICDAR, Vol. 3*. 958–962.
- [33] Zhenzhou Tian, Qinghua Zheng, Ming Fan, Eryue Zhuang, Haijun Wang, and Ting Liu. 2014. DBPD: A Dynamic Birthmark-based Software Plagiarism Detection Tool. In *SEKE*. 740–741.
- [34] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep Learning Similarities from Different Representations of Source Code. In *International Conference on Mining Software Repositories*.
- [35] Chenxi Wang and John Knight. 2001. *A security architecture for survivability mechanisms*. University of Virginia.
- [36] Henry S Warren. 2013. *Hacker's delight*. Pearson Education.
- [37] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 87–98.
- [38] Matthew D Zeiler, Graham W Taylor, and Rob Fergus. 2011. Adaptive deconvolutional networks for mid and high level feature learning. In *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE, 2018–2025.
- [39] Fangfang Zhang, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Program logic based software plagiarism detection. In *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*. IEEE, 66–77.