Identifying Compiler and Optimization Options from Binary Code using Deep Learning Approaches

Davide Pizzolotto
Osaka University
Osaka, Japan
davidepi@ist.osaka-u.ac.jp

Katsuro Inoue
Osaka University
Osaka, Japan
inoue@ist.osaka-u.ac.jp

***Abstract*—When compiling a source file, several flags can be passed to the compiler. These flags, however, can vary between debug and release compilation. In the release compilation, in fact, smaller or faster executables are usually preferred, whereas for a debug one, ease-of-debug is preferred over speed and no optimization is involved. After the compilation, however, most of the flags used cannot be inferred from the compiled file. These flags could be useful in case we want to classify if an older build was made for release or debug purposes, or to check if the file was compiled with flags that could expose vulnerabilities. In this paper we present a deep learning network capable of automatically detecting, with function granularity, the compiler used and the presence of optimization with 99% accuracy. We also analyze the change in accuracy when submitting increasingly shorter amounts of data, from 2048 up to a single byte, obtaining competitive results with less than 100 bytes. We also present our process in the huge dataset creation and manipulation, along with a comparison with other less successful networks using functions of varying size.**

***Index Terms*—Static Analysis, Binary Analysis, Deep Learning, Compilers**

I. Introduction

During the software development life-cycle of a natively compiled application, the process of converting source code to binary code performed by a compiler happens quite often. While performing this transformation, several flags are given to the compiler, signaling the developer's intention to keep or drop some information or to modify the original code in a more optimized version. These flags can be used to optimize towards faster executables, smaller size or lower energy consumption [1]. However, they are not explicitly recorded in the binary file itself as they are completely unnecessary by the machine in order to execute the binary code.

Moreover, also the compiler itself is not easy to identify. There is no standard way to record this information, and although some compilers write a comment in the binary itself, this is easily patchable and not guaranteed to be parsable. In fact, for example, if a file compiled with the clang compiler is linked with a library compiled with gcc, the comment will contain both signatures.

These information, however, are extremely valuable in several applications ranging from categorizing an older build, finding vulnerabilities [2], finding similarities in binaries [3], or providing more accurate bug reports in case the compilation environment can't be controlled [4]. An easy example of the latter case could be a library that exhibit incompatibilities only with a specific compiler, in a product published by a different vendor than the library developer.

Although several works exist in detecting the compiler [4] and toolchain used [5], these methods do not rely on automated learning approaches. Because of this, a notable effort is required to detect the aforementioned information in different architectures, as the new architecture must be studied and understood, in order to check if and how the information can be retrieved. With a machine learning based approach, instead, to detect a new compiler or flag, it is sufficient to provide new data and re-run training.

In this paper we present our approach at recognizing both the compiler and the presence of optimizations using a Long-Short Term Memory network [6] and a Convolutional Neural Network [7]. We analyzed `O0` and `O2` flags on Linux x86_64, compiled by *gcc* or *clang*. Despite not being the first to approach this problem [8], the novelty of our research can be summarized as follow:

- The creation of a huge dataset, with more than 7700 files, compiled with gcc and clang in controlled environments. These environments are compiled from scratch for each compiler-optimization combination. Each generated binary in every environment is used as training data, totaling more than 49GB among all classes.
- The implementation and tuning of a neural network structure that outperforms existing work in flag detection, and is, to our knowledge, the only tool capable of recognizing both optimization flags and compiler.
- An analysis investigating the minimum possible number of raw bytes required in order to get accurate predictions.

Our paper is structured as follows. Section II covers related works in the field of binary file analysis with machine learning. Section III presents the problem and our approach and Section IV its empirical evaluation. Section V compares our choices with another similar work in the field and in Section VI we discuss them in relation to the results obtained. Section VIII, finally, closes the paper. In addition, we put a small Section IX, with pointers to download the replication kit and source code, in order to replicate the evaluation and get the same results.

II. Previous Works

The analysis of binary files is common in the security field. In this field, recently, machine learning techniques have also been used to aid the detection of malware. Pascanu et al. used Recurrent Neural Networks (RNN) [9] to extract malicious features from a binary in an unsupervised way, a work extended by Athiwaratkun et al. with Convolutional Neural Networks [10].

Related works regarding compilers flags, instead, is mostly focused in the effect of these flags rather than their detection. Work performed by Triantafyllis et al. is focused on the exploration of optimal compiler flags [11], as well as the work performed by Hoste et al. [1]. In recent years, on this topic, several machine learning oriented techniques have been developed [12] [13]. Older techniques focused on the usage of machine learning to reduce the number of iterative compilation necessary to get a good set of flags [14] and to help approximate NP-hard problem efficiently, like phase ordering [15]. More recent techniques, instead, uses deep learning to recognize function boundaries, a work done by Bao et al. [16] and then extended by Shin et al. [17]. Chua et al. instead recognized function types using RNNs [18], while He et al. tried to recover the debug symbols from a stripped binary [19].

To our knowledge, the only work trying to detect flags in an existing binary, instead of optimizing them, is the one of Chen et al. [8]. The main differences between theirs and our work are the following:

- We investigate the detection of not only flags but also compilers, in binaries coming from different compilers mixed altogether.
- Our analysis aims at not only maximizing the accuracy, but also minimizing the required input.
- We use a much deeper CNN compared to their studies.

Additional differences and similarities will be discussed after showing the results of our approach, in Section V.

III. APPROACH

The problem we are trying to solve, consists of discerning the optimization level and possibly the original compiler used for the compilation from source code to binary code, given only a portion of the binary code. Specifically, given a sequence of bytes v coming from a binary, we want to train a classifying function $\mathcal{M}$ parameterized by θ_M such that $\mathcal{M}_{\theta_M}(v) = y$ where y is the output prediction. In the first part of the analysis we try to determine only if the binary is optimized or not, thus expecting a value of y in the form $y \in \{0, 1\}$. We refer to this part as the binary classification one. In the second part of the analysis we also try to determine the compiler used, thus our expected value is a number indexing a possible combination compiler-optimization. This second part is referred as the multiclass classification.

It is our goal not only to maximize the accuracy, but also to keep the vector v as small as possible, and as such we dedicate Section III-B to the explanation of how the binary code is transformed into v, the input expected by our learning network. To compare the various performance we train three networks using different models: a Feed-forward Convolutional Neural Network $\mathcal{M}^{\text{CNN}}_{\theta_M}$, a Long-Short Term Memory Network $\mathcal{M}^{\text{LSTM}}_{\theta_M}$and a Dense fully-connected network $\mathcal{M}^{\text{Dense}}_{\theta_M}$, each one with different θ_M parameters. These networks are trained in several different Datasets, explained in detail in Section III-A, and their prediction results compared. More details about the network models can be found in Section III-D.

A. Dataset

In order to train our networks, we need to first collect the data. Our networks perform supervised learning, so it is necessary to have the binary code divided by optimization level and compiler used. Additionally, we want our data to be as much diverse as possible, coming from various computer science topics, projects and written by different people. These requirements allow our networks to not fit only on a particular set of programs but to be applied to a wider range of applications.

Although this task could seem trivial, as plenty of open source software ready to be compiled with the desired flags is available, during the manual compilation several precautions are needed during the linking phase. In fact, despite being capable of deciding both the optimization level and the compilers, we have no guarantees on the environment that will perform the compilation. Several libraries are available to be statically linked, and we don't know anything about the compilation settings that were used for these libraries.

Recall that when a library is statically linked, its binary code is copied inside the final executable during the linking phase. As such, in most build systems, pre-existing libraries could be linked to our fresh controlled build. For this reason, these library could irreversibly contaminate our build given that we lack the information about their generation.

Possible options for solving this problem would be:

1) Use as data only object files prior to linking.
2) Edit the build settings for the generated executable to exclude linking.
3) Build a system from scratch with the desired configuration for every library. Then use this system as the controlled build system.

For the listed options, number 1 would not give us a realistic case study, as several optimizations can be performed also at link-time [20]. Number 2, instead, is "too risky" as it involves checking manually various compilations settings written in different languages and styles in order to ensure dynamic linking and has an high risk of error. For example in our experiments we found that some build scripts use hardcoded parameters, others use environment variables and others recursively integrates files. Checking, understanding and modifying all these build script correctly is definitely not impossible but has a high chance of error.

For this reason we opt for option number 3. Given the huge amount of time required to build an entire system from scratch and the number of possible compiler-flag combinations, we limit our analysis to optimization groups provided by the compilers. These groups are a collection of optimization flags often used together and in our analysis we target levels `O0` and `O2`. These two levels correspond to unoptimized and optimized respectively. During dataset creation we determined that these are the most common configurations, as every open-source software we are using is shipped with one of these two options in the build script. Note that this may not always be the case, as discussed in Section VII

As compilers we use `gcc` version 9.2.0 and `clang` version 10.0.0, two commonly used compilers in Linux environments. We target x86_64 as our only architecture. We follow the *Linux From Scratch* book[1], version 9.1-systemd, published on March 1st, 2020 in order to build the controlled systems. This book is followed because it grants us control on the built system, where each library and binary is ultimately compiled by us with the required compiler and optimization level, without relying on pre-built software.

The particular steps we employ in order to obtain the binaries are the following:

1) We build a toolchain with the required compiler and flags from a host machine, following Chapter 5 of *Linux From Scratch*.
2) We use the toolchain in a chrooted environment to build a clean Linux system, containing software specified in Chapter 6 of *Linux From Scratch*.
3) We use the binaries and libraries composing the system of the previous step as training, validation or testing data.

The first two steps are employed in order to avoid contaminating the build with the host system libraries, and results in the controlled build system. The third step, instead, reflects the fact that the controlled build system can itself be used as dataset.

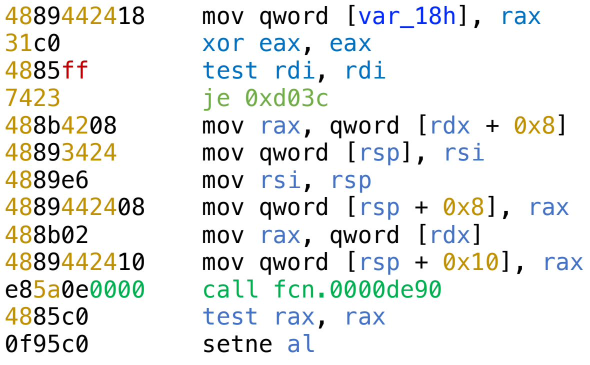
We repeat these steps for the following systems: `gcc-O0`, `gcc-O2`, `clang-O0` and `clang-O2`.

It is worth noting that the GNU C library, `glibc`, fails to compile with anything but optimized gcc, for this reason, we remove every static library built while compiling `glibc` to force the usage of dynamic libraries in that case.

B. Encoding

Possessing the binary files is not sufficient for the analysis, as, in order to submit them to the learning network, it is necessary to convert them into a vector of features v first. We are naturally interested in having a function-grained analysis, as this is the finest grain possible with different compilation options we can expect to encounter in a real case. We propose and compare two approaches: one requiring disassembly and precise function bounds and one using a stream of bytes without prior knowledge about the encoded instructions. The effectiveness of these two approaches is analyzed in Section IV-A.

In the first encoding, the one requiring disassembly, *radare2*[2] is used to extract every function from the executable.



```
4889442418     mov qword [var_18h], rax
31c0           xor eax, eax
4885ff         test rdi, rdi
7423           je 0xd03c
488b4208       mov rax, qword [rdx + 0x8]
48893424       mov qword [rsp], rsi
4889e6         mov rsi, rsp
4889442408     mov qword [rsp + 0x8], rax
488b02         mov rax, qword [rdx]
4889442410     mov qword [rsp + 0x10], rax
e85a0e0000     call fcn.0000de90
4885c0         test rax, rax
0f95c0         setne al
```

Fig. 1. Portion of a disassembled function

The result can be seen in Figure 1. In the Figure, the left column represents the raw bytes written in the binary and the right column their translation in Intel Assembly syntax.

Given that we are working with the x86_64 architecture, we can see a lot of bytes specifying that the registers to be used should be of 64-bit length, represented by the bytes `0x48` in the Figure, preceding every instruction involving `rax`, `rsi`, `rsp` registers. This is a problem, as real functions can be of arbitrary length, but our networks support fixed length vectors as input: we expect these extra bytes add almost nothing to the information and thus decide to strip them and keep only the byte(s) representing the operations to be performed, without any parameter. Unlike the previous research, we do not encode parameters in our representation, in order to save more space and fit even more "valuable" instructions inside the limited length vector [8].

We are thus encoding our functions as time series, where each point in time is actually an opcode of the CPU architecture. For example, the first four instructions of the above image would have this vector: `[0x89,0x31,0x85,0x74]`. In case of multibyte instructions, the multiple bytes are interpreted as multiple values, for example the last two instruction of Figure 1 are composed by the opcodes `0x85` and `0x0F95` but would have this vector: `[0x85, 0x0F, 0x95]`. We choose a-priori 2048 bytes as the maximum length of the input vector v. Extra data is pre-truncated, as we expect the most useful operations to be at the end of a function and not at the beginning which contains initialization, whereas insufficiently long functions are pre-padded with zeroes, as this has been proved to be better for LSTMs [21].

This representation, will be called opcode-based from now on.

The second encoding, is more naive and already proved successful in previous research aimed at learning functions boundaries [16].

In order to generate this second representation, we use `readelf` to dump the `.text` section of the executable and split it into chunks of fixed size. We again choose a-priori 2048 bytes as length of v, however, we evaluate the precision of the networks in recognizing the compilation settings for these chunks varying their size, in order to simulate a real

[1] http://www.linuxfromscratch.org/lfs/index.html

[2] https://rada.re/

```
f0 25 14 de af 8c 85 c3    00 f0 25 14 de af 8c 85
85 bf 5b cf e0 f2 63 0b    00 00 00 00 85 bf 5b cf
92 af 97 0b 06 84 1d 5d    00 00 00 92 af 97 0b 06
e3 14 bc ac a8 de 21 e7    00 00 00 00 00 00 e3 14
73 11 27 9a ff 4f d9 73    00 00 00 00 00 73 11 27
03 d6 ce de 8b 0d af 46    00 00 03 d6 ce de 8b 0d
74 37 35 f2 49 c3 e5 69    00 00 00 74 37 35 f2 49
8c 47 4a 57 d2 cf 7e 46    00 8c 47 4a 57 d2 cf 7e
```

Fig. 2. Truncation of input sequences on the left and subsequent padding on the right

case where functions can have different length.

A drawback of this second representation is that we are completely unaware if the raw data represent instructions or stack data. On the other hand, disassembly is not required, a step that usually requires several minutes and is necessary for the first representation presented and the previous research [8].

C. Padding

For the first encoding presented in Section III-B, the input vector length is the same as the function length. This makes the previously described padding necessary, as the function length is always different. On the other hand, the second approach, requires only a fixed amount of sequential data from the binary, thus needing no padding.

However, we determined that by providing always chunks in a fixed size, both CNN and LSTM are unable to deal with padding during evaluation. Experimental data shows that when training with fixed size chunks, the inference of a chunk padded with zeroes by more than 60% of its length results in an unacceptable drop below 80% accuracy. This can be detrimental in a real case, as it would require training different models for different input sizes in case we want to infer smaller amount of data or just a portion of the executable.

To fix this problem, we truncate the chunk to a random length in the interval $[32, \textit{input length}]$. Then we pre-pad its length in the same manner as the first approach described in Section III-B. The value 32 has been chosen in order for the chunk to be still classifiable: padding too much could leave us with chunks where the classification is impossible due to lack of enough information.

An example of this can be seen in Figure 2: each line represents an input sequence before padding on the left block and after padding on the right block. The red part is the amount of input data that will be truncated. The length of this part is decided randomly within interval bounds mentioned previously in the section. On the right block we can see that the same amount is replaced by prepending zeroes.

Training the networks with chunks modified in this way allow to infer, with more than 90% accuracy, sequences composed up to 99.5% by zeroes. A more in-depth evaluation of this padding is provided in Section IV-D.

D. Networks

For our analysis we use three different networks modeled as follows: a Fully Connected Dense Network, a Long-Short Term Memory (LSTM) [6] and a Feed Forward Convolutional Neural Network (CNN) [7]. The last two networks have been chosen due to their successful applications in Natural Language Processing or Image Recognition. We are thus modeling our problem of optimization detection into a pattern recognition problem: a particular optimization can be recognized by a network as a pattern in the input vector. Small perturbations have been applied to the original layers structure to determine the best configuration for our problem.

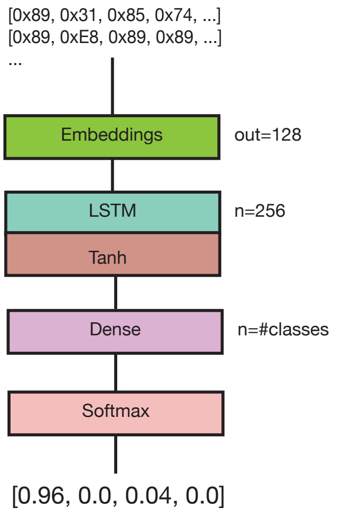


Fig. 3. LSTM model structure

The first network, Dense, is not presented in detail as it is use merely as a comparison to show its inefficiency. It is composed of three fully connected dense layers of respectively 2048, 1024 and 512 units. The last layer, responsible of the prediction is made of 1 unit for the binary case and 4 units for the multiclass. After each layer ReLU is used as activation function and right after, a Dropout of 20%. The optimizer used is Adam with learning rate 10^{-3} and the activation function of the final layer is a Sigmoid for the binary case and a Softmax with four classes for the multiclass one.

The second model is depicted in Figure 3. This model implements a simple LSTM, given that we encoded our sequence of bytes as a time series and the ability of LSTMs to perform well in that kind of problems. LSTMs in fact have special "memory" cells, that can be used to memorize a particular input or pattern even in long sequences [6]. Our core idea resides in training this kind of model into memorizing a particular pattern, representing the compiler or the optimization level, over a long sequence of bytes belonging to the binary.

As we can see from the picture the model is pretty straightforward, composed firstly by an embedding layer with 256 as vocabulary size (as we are using bytes in range 0x0 to 0xFF) and 128 as dimension for the dense embedding. This layer encodes positive integers into a dense vector of fixed size, understandable by the LSTM. Then the LSTM layer with

256 units is used for the actual learning. This layer uses an hyperbolic tangent (tanh) as activation function. The kernel is initialized by drawing samples from an uniform distribution in $[-64^{-\frac{1}{2}}, 64^{-\frac{1}{2}}]$. The last part of the network is a Dense with 1 output and Sigmoid activation for the binary case, Dense with 4 outputs and Softmax for the multiclass case. The optimizer used is again Adam with learning rate 10^{-3}.

The last model, comes from the trend in the Image recognition and categorization [22] and is based on a Convolutional Neural Network. The idea is that a set of convolutions is used to extract highly dimensional information from the sequence of raw bytes passed as input. We can see the structure in Figure 4.

The first layer is identical to the one in the LSTM version, as its utility is the same. Then three blocks of convolution, convolution, pooling are used, with increasing number of filters. In the Figure the label k3n32s1 for a convolution layer indicates: kernel size 3, number of filters 32, strides 1. In these blocks the convolutions are used to extract features from the sequence of bytes, and the pooling is used to make these features independent of their position in the sequence.

The Leaky ReLU [23] is used in place of the ReLU [24] as the latter suffer from the vanishing gradient problem. In fact while the ReLU function returns 0 for values less than 0, the leaky variant returns an ϵ, in our case 0.01, in order to keep the neuron still alive. Given that, as explained in Section III-B, we are using highly padded sequences, we use the leaky version to avoid the gradient vanishing and remaining at zero for the rest of the training. Before the output, one last fully connected layer composed of 1024 neurons is used, followed by a ReLU activation and the canonical Dense and Sigmoid for binary classification or Dense and Softmax for multiclass classification. Also in this case the optimizer is Adam with learning rate 10^{-3}.

All the models use Binary Cross-entropy as loss function for the binary classification and Categorical Cross-entropy for the multiclass classification [25].

The presented hyperparameters of both LSTM and CNN have been estimated using the Hyperband algorithm [26]. We used power of two as space search in the interval $[32, 1024]$ for most features, except kernel size and strides. For the kernel size the space search was the set $\{3, 5, 7\}$. The space search for the stride value was instead the set $\{1, 2\}$.

IV. Evaluation

We performed experiments using an Nvidia Quadro RTX5000 on the data presented in Section III-A after the preprocessing explained in Section III-B. However, given the high amount of binary data, we obtained a number of input vector in the order of millions. We could thus safely split data into disjoint set, while keeping a high number of samples for each set. These sets are training, validation and testing with a split ratio of 50%, 25% and 25% respectively.

No augmentation has been performed and no overlapping sequences were collected, thus every sample was absolutely unique between training validation and testing. Moreover, duplicated samples (for example system calls) were removed from each set. No functions have been removed in case they were too small. Each class has then been balanced by randomly removing data not only in the training, but also in the validation and testing set. After this preprocessing operation, the number of samples used can be seen in Table I

TABLE I
Number of samples for each configuration

Dataset	Training	Validation	Testing
$\mathcal{D}_{func}$	44876	22438	22438
$\mathcal{D}_{gcc}$	46378	23190	23188
$\mathcal{D}_{clang}$	388372	194186	194186
$\mathcal{D}_{mixed}$	92746	46374	46374
$\mathcal{D}_{multi}$	92756	46380	46376

In the Table the dataset names are the following:

$\mathcal{D}_{func}$
Dataset composed by extracted opcodes as described in Section III-B. It is composed by programs composing the Linux From Scratch build, coming from `gcc-O0` and `gcc-O2`. Binary classification.

$\mathcal{D}_{gcc}$
Dataset composed by raw values from programs composing the Linux From Scratch build, coming from `gcc-O0` and `gcc-O2`. Binary classification.

$\mathcal{D}_{clang}$
Dataset composed by raw values from programs composing the Linux From Scratch build, coming from `clang-O0` and `clang-O2`. Binary classification.

$\mathcal{D}_{mixed}$
Dataset composed by raw values from programs coming from all built Linux From Scratch images. `gcc-O0` and `clang-O0` are merged into a single class, as well as `gcc-O2` and `clang-O2`. In this case also the various samples coming from `gcc` and `clang` has been balanced inside the `O0` and `O2` classes. Binary classification.

$\mathcal{D}_{multi}$
Dataset composed by raw values from programs coming from all the configurations. Unlike $\mathcal{D}_{mixed}$, every class is independent. Multiclass classification.

Training has been performed for 40 epochs using batch sizes of 256 samples. An early stopper was employed, stopping the learning after three epochs without at least an improvement of 10^{-3} in the loss function on the validation dataset. Additional info about the training process are provided in the replication kit in Section IX. Table II shows the time required for each sample during training and inference.

In this section we want to answer the following research questions:

- **RQ**$_{func}$: Is it better to use the opcodes encoding or just feed raw bytes?
- **RQ**$_{binary}$: How the various models perform while detecting the optimization level?

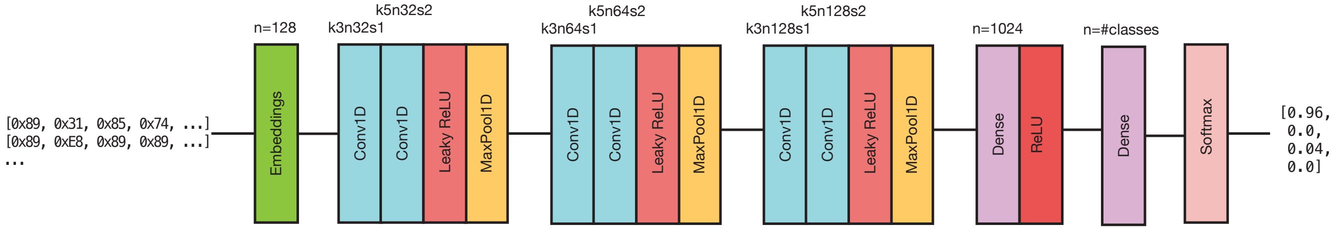


Fig. 4. CNN model structure

TABLE II
TIME REQUIRED TO PROCESS EACH SAMPLE IN MICROSECONDS

Model	Training (μs)	Inference (μs)
Dense	65	20
LSTM	9000	2000
CNN	850	230

- **RQ**$_{mixed}$: How the various models perform while detecting the optimization level if multiple compilers are mixed?
- **RQ**$_{multiclass}$: How the various models perform while detecting both the optimization level and the compiler used?
- **RQ**$_{pad}$: Does padding during training improve the performance of the networks?

In order to answer these questions we perform the evaluation for increasingly padded values of v. This means that we evaluate the performance of the model when submitting only inputs composed by 1 bytes, 2 byte, and so on until the 2048 bytes used as limit for the input vector. In this way we can get an insight on how the model would perform in case less than 2048 bytes are available.

To better compare the obtained results with the average case, we present the decompilation results of three files, a small, medium and huge sized, coming from the `clang-O0` compilation in Table III.

TABLE III
STATISTICS ABOUT A SMALL, MEDIUM AND HUGE FILE, WITH MINIMUM, MAXIMUM AND AVERAGE FUNCTION LENGTH IN BYTES

Filename	Size (bytes)	Functions	Min	Max	Average
dirname	$9.4 \cdot 10^3$	13	6	492	55.07
ninja	$4.5 \cdot 10^6$	4039	1	4747	82.78
clang-10	$1.48 \cdot 10^9$	722849	1	53705	133.16

A. Encoding

In order to answer RQ$_{func}$, we train an LSTM over two different datasets: one on $\mathcal{D}_{func}$ and one on $\mathcal{D}_{gcc}$. As previously explained, the first of these datasets contains only the opcode composing the functions, where the second contains raw bytes of data. We perform the evaluation by classifying the various samples in the testing set with progressively increasing bytes. The plot in Figure 5 shows the results.

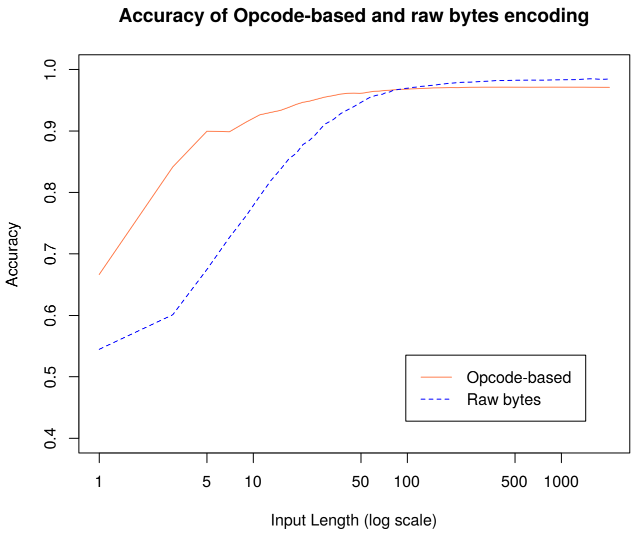


Fig. 5. Accuracy of the opcode encoding compared to raw encoding

We can see that the opcode based encoding has better accuracy for very short sequences of bytes, then the raw bytes encoding matches and eventually performs better than the opcode based for longer sequences of more than 300 bytes. Despite the opcode-based being superior in matter of information carried per single byte, note that for each opcode we usually have more than one raw byte. This can be easily seen in Figure 1, where the first four instructions are converted into four bytes with this encoding, but their raw equivalent would be a total of 12 bytes. Unfortunately, given the varying nature of each instruction in x86_64 it is not possible to plot a precise comparison of information available, rather than information carried. The idea of the opcode-based encoding, as explained in Section III-B was to squeeze as much high-valued information as possible into the input vector v in case of longer function, however, results show that for longer sequences the raw bytes encoding performs better. For very short sequences of instruction, although the function based encoding performing better on a per-byte basis usually the amount of raw bytes is much higher thus still allowing a more

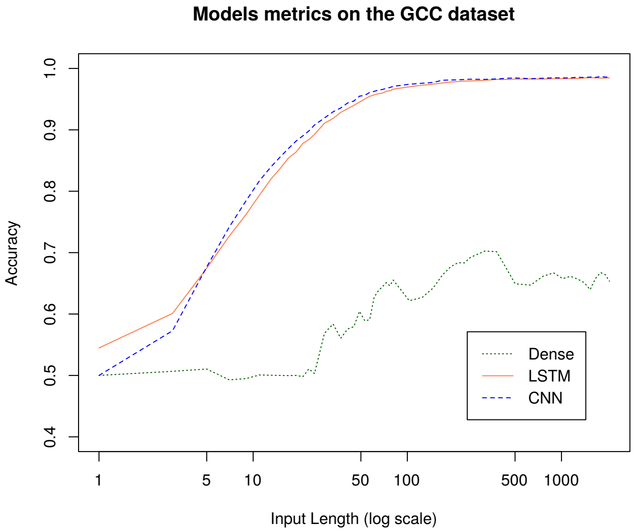


Fig. 6. Accuracy of the various model in the $\mathcal{D}_{gcc}$ dataset.

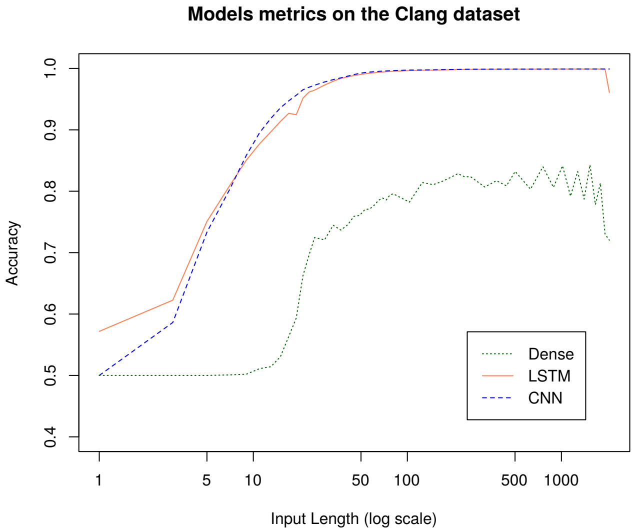


Fig. 7. Accuracy of the various model in the $\mathcal{D}_{clang}$ dataset.

accurate prediction. We can thus answer RQ$_{func}$ as follow:

The opcode-based encoding grants a much higher accuracy per-byte. However, usually, the availability of raw bytes is numerically superior, allowing for better comparisons

Given these results, following experiments are performed and reported only with the raw bytes encoding.

B. Binary classification performance

In order to decide which model performs better between the Dense, LSTM and CNN ones, answering RQ$_{binary}$, we perform the comparison by training each of them in two different datasets. The first dataset is $\mathcal{D}_{gcc}$, containing data obtained by binaries compiled with `gcc` and split into `O0` and `O2` classes. Figure 6 shows the results.

In the Figure we can easily note the inability of the Dense network at recognizing the optimizations, however, regarding the LSTM and CNN, their performance is essentially identical, with the latter being slightly better. A similar situation can be seen in Figure 7 where the same training has been performed but with $\mathcal{D}_{clang}$, the dataset containing data obtained from binaries compiled with `clang`.

Interestingly enough, in this plot the Dense network performs much better: despite the results being not even comparable with the LSTM and CNN ones, a dataset almost ten times bigger shows the Dense network ability to learn something. To conclude, we also present precision, recall and F1-score of the models in the $\mathcal{D}_{gcc}$ dataset, with the $\mathcal{D}_{clang}$ providing similar results for these two models. These can be found in Figure 8.

Table IV instead presents the metric values for the $\mathcal{D}_{gcc}$ dataset with a v length of 2048.

Recall, however, that despite the similar performance between LSTM and CNN, the former requires much more training and inference time, as shown in Table II. We can thus answer RQ$_{binary}$ as follow:

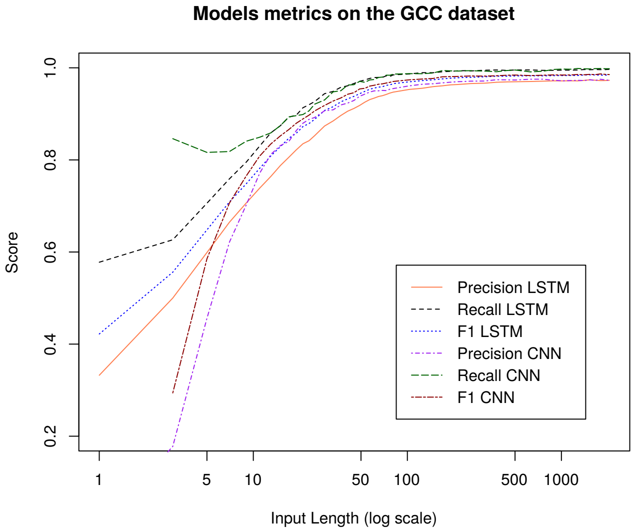


Fig. 8. Precision, Recall and F1 of the various model in the $\mathcal{D}_{gcc}$ dataset.

TABLE IV
METRICS OBTAINED IN THE GCC DATASET WITH THE LONGEST INPUTS

Model	Accuracy	Precision	Recall	F1
LSTM	98.48%	97.28%	99.67%	98.46%
CNN	98.55%	97.29%	99.80%	98.53%

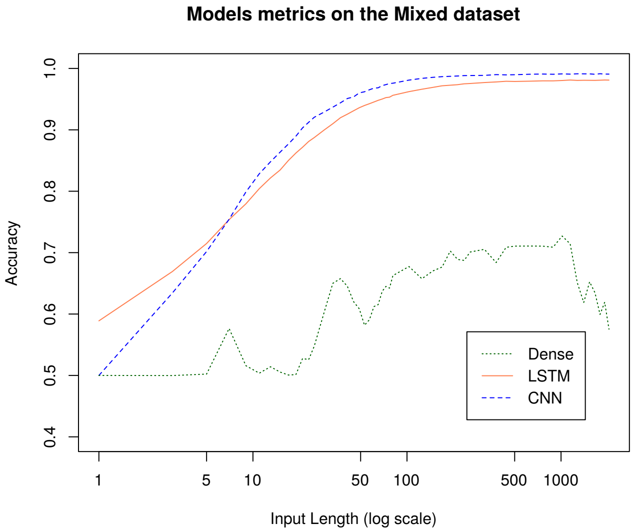


Fig. 9. Accuracy of the various model in the $\mathcal{D}_{mixed}$ dataset.

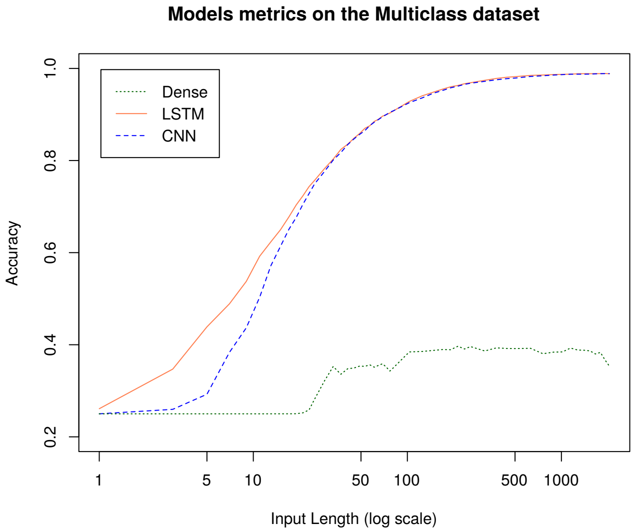


Fig. 10. Accuracy of the various model in the $\mathcal{D}_{multi}$ dataset.

Both LSTM and CNN have an almost identical performance related to accuracy, precision, recall and F1-score. The LSTM however is almost ten times slower at training and at inference time. The Dense model instead is not able to reach an acceptable accuracy in any case.

These results proves that both the LSTM and CNN models can recognize if a portion of binary is optimized or not, however, they require to be trained in the specific compiler they were designed to recognize. This implies that before recognizing the optimization is necessary to recognize the compiler in order to choose the correct inference network.

For this reason, wanting to avoid this, we perform the evaluation of RQ$_{mixed}$ by mixing altogether samples taken from $\mathcal{D}_{gcc}$ with samples taken from $\mathcal{D}_{clang}$, while keeping them separated by optimization level. The result, $\mathcal{D}_{mixed}$, is used to train the next set of Dense, LSTM and CNN networks that should correctly recognize the optimization without being trained specifically for a given compiler. Lastly, note that $\mathcal{D}_{clang}$ has a much higher number of samples than $\mathcal{D}_{gcc}$. In order to get correct results we also balance the number of samples for each compiler inside each predictable class, in training, validation and testing. After training each model for this dataset, the accuracy values are shown in Figure 9.

This results are essentially unchanged from the compiler-specific training, however, the CNN performs even more better than the LSTM. Precise results are showed, for 2048 bytes, in Table V.

TABLE V
METRICS OBTAINED IN THE MIXED DATASET WITH THE LONGEST INPUTS.

Model	Accuracy	Precision	Recall	F1
LSTM	98.10%	96.50%	99.68%	98.06%
CNN	99.07%	98.34%	99.79%	99.06%

At this point we can answer also RQ$_{mixed}$ as follow:

CNN performs better than LSTM in the mixed dataset and is faster at both training and inference time. The Dense model still has no acceptable accuracy

C. Multiclass classification performance

Despite both CNN, and LSTM being able to correctly detect optimization levels in binary data coming from different compilers, the compiler is still not considered in the prediction. In this section we evaluate the performance of the three models while predicting not only if the binary is optimized or not, but also the compiler that generated it. As explained in Section III-D, the networks are slightly different in the last Dense layer and activation function, compared to the one used for binary classification. The dataset used is $\mathcal{D}_{multi}$. Figure 10 shows the accuracy of the prediction for the various models. In case the compiler was predicted correctly, but with the wrong optimization level we still consider the result a wrong prediction while computing the accuracy.

We can see that also this case reflects the binary classification: the Dense model is unable to learn anything while LSTM and CNN models have comparable result. Interestingly, however, this time the LSTM performs better than the CNN for the first 100 bytes, instead of the usual 10 in the binary classification, and in the end the CNN has much less advantage in accuracy over the LSTM. Moreover, if in binary classification 50 bytes are required to achieve 90% accuracy, in the multiclass double the amount is necessary to have the same result.

Table VI shows the accuracy for the 125 and 2048 bytes vector.

In our experiment, however, the LSTM proved to be really difficult to train in this case as we managed to obtain this result at the fourth attempt in training. In previous attempts,

TABLE VI
ACCURACY FOR THE MULTICLASS CLASSIFICATION USING DIFFERENT BYTES INPUT.

Model	Accuracy (125)	Accuracy (2048)
LSTM	94.05%	98.84%
CNN	93.61%	98.87%

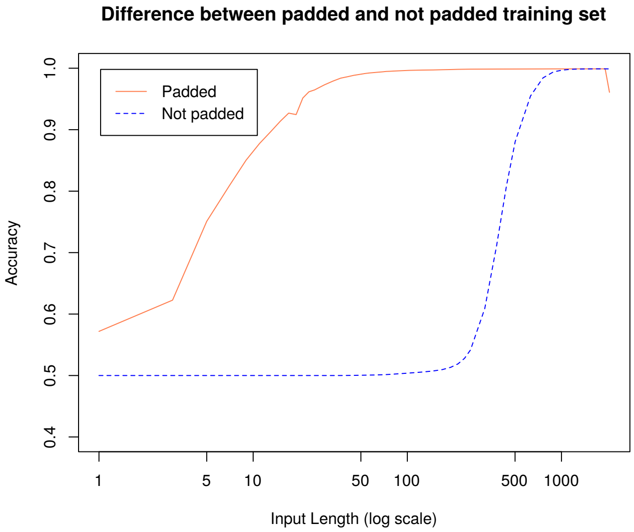


Fig. 11. Accuracy of the LSTM model if it has been trained with padded values or not

the LSTM topped respectively at 63% accuracy, 76% accuracy and 85% accuracy with the highest length vector.

We can thus answer RQ$_{multiclass}$ as follow:

In the multiclass classification CNN and LSTM provides similar accuracy, with the latter being slightly better with input smaller than 100 bytes. The LSTM is however harder and slower to train. The Dense model still provides unacceptable accuracy.

D. Padding

In Section III-B we assert that our networks perform worse if, during training, raw bytes sequences are never padded, and then padded sequences are predicted. In this section we present RQ$_{pad}$, investigating the difference between padding in training or not. In this experiment we use the same network with the same dataset, seed, and samples ordered in the same way, except that in one case the input values are always of 2048 bytes, in the other they are randomly cut in the interval $[32, 2048]$ and padded with zeroes. Then we perform the evaluation in the same way of the other Research Questions. We evaluate the LSTM architecture on the biggest dataset we have, $\mathcal{D}_{clang}$, obtaining the results showed in Figure 11.

By further analyzing the confusion matrix, we can tell that the LSTM trained without padded values always predicts “not optimized” for inputs shorter than 235 bytes and the accuracy matches the LSTM trained with padded values only at around 1500 bytes, with only 30% of the total sequence padded.

Although not formally presented, the CNN architecture suffered the same problem in the multiclass dataset, predicting the same class for inputs shorter than 140 bytes and then matching the padded-trained counterpart at around 1000 bytes. We can thus answer RQ$_{pad}$ as follow:

If inputs are always of the same size during training, networks will have significantly lower accuracy in predicting shorter sequences.

To summarize the five Research Questions we can conclude the following:

Although the LSTM providing better accuracy for very small inputs, the CNN is faster and easier to train. For input sizes where the CNN is outperformed by the LSTM, the accuracy is still low even in the LSTM model, making the CNN a preferable choice in any case.

V. COMPARISON

Given the similarities with the work of Chen et al. [8], the only related work attempting to recognize compiler flags in a binary file, in this section we report the comparison with their work along with a reasoning about the differences.

One of the main difference is the dataset used. In our study we needed additional binaries coming from clang and, especially, we wanted to be absolutely certain that static libraries could not interfere with the flags or the compiler used. For this reason we limited the analysis to only two optimization flags, given that replicating their study would mean building at least 10 different Linux systems from scratch. Moreover, our dataset is huge compared to their work. Only one executable out of the 7701 we used, `clang-10`, is composed by 722804 functions, almost double their entire amount of functions for all the categories combined. This can be easily explained as we used *radare2* instead of *objdump* as disassembler, a more sophisticate tool that requires much more time but can also detect indirect functions.

Despite disassembling the executable, in order to compare the opcode-based approach with the raw bytes one, this task is essentially not needed in our approach, given that we preferred the raw bytes one. Using a more sophisticated disassembler in order to get more functions can be very time consuming: in our machine for example disassembling `bison`, a 400KiB executable, takes around 10 seconds. Disassembling `busybox` instead, a 2MiB executable, takes 66 seconds. For larger binaries, if the function grain is not required, their approach still requires function beginning and ending, where ours can just take a random 2048 bytes from the `.text` section and be dominated by the inference time.

In addition, unlike them, we did not remove short functions from the dataset: doing so would create differences between evaluation and a real case study. On the contrary, we took this as a challenge and tried to understand the fewest amount of bytes required to get an accurate prediction. As shown in the evaluation, we got a 90% accuracy, similar to their shallow model, with only 45 bytes of input in the binary case, and around 80 in the multiclass one. Assuming a mean of 4 bytes for each instruction, we can see that removing short functions is not recommended.

About the results, in the previous research the authors claim that their study "exhibits accuracy of around 89%". Althuogh our approach can reach much higher accuracy, we expect having to classify more than just two flags to be harder. In the previous work, the authors also claimed that the GRU [27] model, a kind of RNN, achieved much higher accuracy than the CNN, although here we proved that CNN can be comparable, if not better than RNNs (as LSTM is a kind of RNN), even in multiclass classification, provided the CNN is deep enough. In fact we used 6 convolution layers, 3 pooling layers and 2 dense layers, where the original paper claimed to have tried CNN with two layers.

In the end, the two analyses can not be compared one-to-one, as the previous one is focused more in the number of optimization flags while we focused on the combinations flags-compiler. For this reason, we plan to extend our analysis also to the remaining flags in order to have a full evaluation and comparison.

VI. Discussion

Results obtained in Section IV provides, as an implication, the possibility of detecting the binary flags with multiple granularity in a fast way. In fact, we showed the possibility to recover, with high accuracy, the compiler and optimization flags used, with an inference time of $230\mu s$, as showed in Table II. In addition to the fact that dumping raw bytes from an executable or library is really fast, as it involves just reading the file itself, we can detect with very high confidence, reaching almost 99%, the compilation settings with a file-grain in less than one millisecond.

If a function grain is necessary, with our method disassembling is still not necessary, as the functions headers can be retrieved by other means (i.e. Deep Learning) and thus avoiding again the slowest part of binary analysis. This allows our tool to be used to check, even at run-time without a noticeable performance impact, if the flags used for compilations were unsafe, energy oriented, speed oriented, or forgotten at all.

In addition, the tiny amount of bytes required by our method allow us to target very small portions of code, and thus can be used to check which portions of the binary matches the used compilation flags. This fact allows a better categorization of the binary content and could help in binary analysis. In fact, if a small portion of file with different flags and maybe compiler than the rest of the file is found, there is a high likelihood that this portion belongs to a static library. For this reason, that part can be, not skipped as it could be a file purposely compiled with different flags, but analyzed with lower priority.

VII. Limitations and Future Works

The analysis we conducted was limited to a single architecture, a pair of compilers and a pair or optimization levels. As we expect the current model, accordingly retrained, to work also with more compilers and in different architectures we have no guarantees that this would be the case. A threatening hypothesis is that our network could learn particular optimization patterns generated by compilers only in the x86_64 architecture that maybe are harder to learn in other architectures. For this reason, we expect to conduct a similar study also in other architectures such as ARM. This last architecture, for example, has a fixed length encoding, the lack of which was a problem in the analysis explained in Section IV-A that compares opcode-encoding and raw data.

Moreover, the difference between `O0` and `O2` is pretty huge compared to the difference between `O2` and `O3`. Future works will involve performing multiclass classification even with the flags, by checking also other common settings like the "minimum size executable". It would be interesting also to check the classification performance with a multilabel classification where each label is a different compilation flag. However this probably would be unfeasible in the number of required data to be generated, at least with our method that requires manually compiling an entire Linux system for each class.

VIII. Conclusion

In this paper we described two deep learning networks, one based on an Long-Short Term Memory model and the other based on a Convolutional Neural Network model. We evaluated them and showed that they both can achieve above 98% accuracy in both optimization and compiler detection, with peaks of more than 99% when more data is available.

We also showed that even with very short input sequences of just 125 bytes, corresponding to roughly 30 instructions, the accuracy of the networks is around 95%. This enables the classification not only at executable grain, but even at function grain, given that we saw in Table III the average function size is around 130 bytes. In case the function grained evaluation is not needed, disassembly, an time-expensive operation, is not required as we showed that both networks perform equally good while handling inputs composed of raw bytes.

IX. Replication

A replication kit can be found on Zenodo at the following url [28]. The kit contains source code, dataset and models. Instructions on how to replicate the results can be found in the contained README.

The source code can also be found publicly on GitHub: `github.com/inoueke-n/optimization-detector`.

Acknowledgements

This work was supported by JSPS KAKENHI Grant Number 18H04094.

References

[1] K. Hoste and L. Eeckhout, “Cole: compiler optimization level exploration,” in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, 2008, pp. 165–174.

[2] M. Demertzi, M. Annavaram, and M. Hall, “Analyzing the effects of compiler optimizations on application reliability,” in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, 2011, pp. 184–193.

[3] Y. David, N. Partush, and E. Yahav, “Similarity of binaries through re-optimization,” in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 79–94.

[4] N. E. Rosenblum, B. P. Miller, and X. Zhu, “Extracting compiler provenance from program binaries,” in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2010, pp. 21–28.

[5] N. Rosenblum, B. P. Miller, and X. Zhu, “Recovering the toolchain provenance of binary code,” in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011, pp. 100–110.

[6] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, pp. 1735–80, 12 1997.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[8] Y. Chen, Z. Shi, H. Li, W. Zhao, Y. Liu, and Y. Qiao, “Himalia: Recovering compiler optimization levels from binaries by deep learning,” in *Intelligent Systems and Applications*, K. Arai, S. Kapoor, and R. Bhatia, Eds. Cham: Springer International Publishing, 2019, pp. 35–47.

[9] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas, “Malware classification with recurrent networks,” in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2015, pp. 1916–1920.

[10] B. Athiwaratkun and J. W. Stokes, “Malware classification with lstm and gru language models and a character-level cnn,” in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2017, pp. 2482–2486.

[11] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, “Compiler optimization-space exploration,” in *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 2003, pp. 204–215.

[12] Z. Wang and M. O’Boyle, “Machine learning in compiler optimization,” *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1879–1901, 2018.

[13] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, “A survey on compiler autotuning using machine learning,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–42, 2018.

[22] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.

[14] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. O’Boyle, J. Thomson, M. Toussaint, and C. K. Williams, “Using machine learning to focus iterative optimization,” in *International Symposium on Code Generation and Optimization (CGO’06)*. IEEE, 2006, pp. 11–pp.

[15] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O’Reilly, “Meta optimization: Improving compiler heuristics with machine learning,” *ACM sigplan notices*, vol. 38, no. 5, pp. 77–90, 2003.

[16] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, “{BYTEWEIGHT}: Learning to recognize functions in binary code,” in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 845–860.

[17] E. C. R. Shin, D. Song, and R. Moazzezi, “Recognizing functions in binaries with neural networks,” in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 611–626.

[18] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, “Neural nets can learn function type signatures from binaries,” in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 99–116.

[19] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, “Debin: Predicting debug information in stripped binaries,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1667–1680.

[20] A. V. Aho, R. Sethi, and J. D. Ullman, “Compilers, principles, techniques,” *Addison wesley*, vol. 7, no. 8, p. 9, 1986.

[21] M. Dwarampudi and N. V. S. Reddy, “Effects of padding on lstms and cnns,” *CoRR*, vol. abs/1903.07288, 2019. [Online]. Available: http://arxiv.org/abs/1903.07288

[23] A. L. Maas, A. Y. Hannun, and A. Y. Ng, “Rectifier nonlinearities improve neural network acoustic models,” in *Proc. icml*, vol. 30, no. 1, 2013, p. 3.

[24] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,” in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[25] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[26] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, “Hyperband: A novel bandit-based approach to hyperparameter optimization,” *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6765–6816, 2017.

[27] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using rnn encoder-decoder for statistical machine translation,” *arXiv preprint arXiv:1406.1078*, 2014.

[28] D. Pizzolotto and K. Inoue, “Compiler flag detection using cnn,” May 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3865122