Deep Learning Approaches for Identifying Compiler and Optimization Options from Binary Code.

Mistry Unnat, 20BCE515
Student
Computer Science and Engineering Department
Nirma University, A'bad, India

Neel Prajapati, 20BCE518
Student
Computer Science and Engineering Department
Nirma University, A'bad, India

Pratyush Khare, 20BCE519
Student
Computer Science and Engineering Department
Nirma University, A'bad, India

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.**

I. Introduction

A compiler's conversion of source code to binary code occurs frequently during the software development life-cycle of a natively built programmer. Several flags are passed to the compiler during this transformation, indicating the developer's purpose to maintain or remove certain information or to change the original code into a more streamlined version. With the use of these flags, an application may be optimised for quicker executables, smaller sizes, or less energy usage. However, because the computer doesn't require them at all to run the binary code, they aren't explicitly stored in the binary file itself.

Furthermore, it might be challenging to pinpoint the compiler itself. Although some compilers add a note in the binary itself, this is easily patchable and is not always guaranteed to be parsable. There is no standard mechanism to store this information. In fact, the remark will include both signatures, for instance, if a file built with the clang compiler is linked with a library built with the gcc compiler.

However, these details are very useful for a variety of purposes, such as classifying an earlier build, identifying vulnerabilities, identifying commonalities across binaries, or generating more accurate bug reports if the compilation environment cannot be controlled. A simple illustration of the latter scenario would be a library that only displays compatibility issues with a certain compiler in software developed by a different vendor than the library provider.

Despite the fact that there have been several attempts to identify the compiler and toolchain used, these techniques do not rely on automatic learning techniques. Since the new design must be examined and comprehended in order to determine if and how the information may be collected, a significant amount of work is needed to detect the aforementioned information in different architectures. Instead, using a machine learning-based technique, it is simple to supply fresh data and re-run training to identify a new compiler or flag.

We describe in this study how we use a Long Short Term Memory network and a Convolutional Neural Network to identify the compiler and the presence of optimizations. On Linux x86 64, built by gcc or clang, we examined the O0 and O2 options. Our research's uniqueness, although not being the first to tackle this issue, may be summed up as follows:

- The construction of a sizable dataset with more than 7700 files that was assembled under controlled conditions using clang and gcc. Each compiler-optimization combination necessitates the compilation of these environments from scratch. In all environments, every created binary is used as training data, amounting to more than 49GB across all classes.
- The development and improvement of a neural network structure that, to our knowledge, is the only tool able to recognise both compiler and optimization flags, outperforming previous work in flag identification.
- A study examining the bare minimal amount of raw data that can be used to provide reliable forecasts.

II. APPROACH

Finding the optimization level and, presumably, the original compiler used to convert source code to binary code from a small sample of the binary code is the issue we are attempting to resolve. We wish to train a classifying function M parameterized by M such that MM(v) = y, where y is the output prediction, given a series of bytes v originating from a binary. We only attempt to ascertain if the binary is optimised or not in the first section of the analysis, therefore we anticipate a value of y in the range of 0 to 1. This portion is referred to as the binary classification portion. Our predicted value is a number that indexes a potential combination of compiler-optimization since we also attempt to identify the compiler in the second round of the investigation. The multiclass classification is the name given to this second section.

We focus Section III-B on describing how the binary code is converted into v, the input that our learning network expects, because it is not just our aim to optimize accuracy but also to make the vector v as short as feasible. We train three networks with distinct models—a Feed-forward Convolutional Neural Network (CNN), a Long-Short Term Memory Network (LSTM), and a Dense Fully Connected Network (Dense)—each with a different set of M parameters—to compare the diverse performance.

A. *Dataset*

We need to get the data before we can train our networks. Because supervised learning is what our networks do, it is essential to separate the binary code by compiler and optimization level. Furthermore, we want our data to be as diversified as possible, drawn from a range of computer science-related subjects, projects, and authors. These specifications enable our networks to be used for a greater variety of applications rather than just a certain set of programmers.

Although this operation could appear simple given the abundance of open-source software that is already prepared to be built with the relevant flags is accessible, many safety considerations are required during the linking step of hand compilation. In actuality, we have no promises about the environment in which the compilation will take place, despite our ability to choose the optimization level and the compilers. There are a number of libraries that can be statically linked, but we have no idea what compilation options were used for them.

Remember that during the linking process, the binary code of a library that is statically linked gets copied into the final executable. As a result, old libraries may be connected to our new controlled build in the majority of build systems. Due to the fact that we are unaware of the circumstances behind these libraries' creation, they may taint our construct in an irreversible manner.
Possible solutions to this issue include:
1) Prior to linking, use only object files as data.
2) To disable linking, modify the build parameters for the executable that is produced.
3. Create a brand-new system with the ideal setup for each library. Use this system as the controlled build system from that point forward.

For the listed options, Number 1 would not provide us with a meaningful case study for the choices provided because various optimizations can also be carried out at link-time. Number 2 is "very dangerous" in contrast since it entails manually reviewing several compilation parameters for different languages and writing styles in order to achieve dynamic linking and has a high mistake rate. For instance, during our tests, we discovered that some build scripts make use of hardcoded parameters, others use environment variables, and still others of recursive file integration. It is certainly feasible to check, comprehend, and adjust each build script appropriately, but there is a significant risk of mistake.

Due to this, we select option number 3. We restrict our research to the optimization groups offered by the compilers because to the enormous amount of time needed to create a full system from scratch and the variety of compiler-flag combinations that are conceivable. In our investigation, we focus on levels O0 and O2, and these groups are a collection of optimization flags that are frequently used together. These two tiers represent, respectively, unoptimized and optimised. Due to the fact that every piece of open-source software we use ships with one of these two choices in the build script, we were able to discover that these configurations are the most typical during the dataset creation process.

We employ gcc version 9.2.0 and clang version 10.0.0, two compilers that are often used in Linux settings.

We solely aim for the x86 64 architecture. To create the controlled systems, we adhere to the instructions in the Linux From Scratch book, version 9.1-systemd, released on March 1st, 2020. This book is used because it gives us control over the constructed system and eliminates the need for pre-made software by allowing us to build each library and binary using the appropriate compiler and optimization level.

The specific procedures we use to get the binaries are as follows:

1. In accordance with Chapter 5 of Linux From Scratch, we create a toolchain from a host system using the necessary compiler and options.
2. We create a clean Linux system using the software listed in Chapter 6 of Linux From Scratch using the toolchain in a chrooted environment.
3. As training, validation, or testing data, we employ the binaries and libraries that make up the system in the previous phase.

The first two stages are used to create the controlled build system and prevent host system libraries from contaminating the build. The controlled construction system itself may be utilised as a dataset, as shown in the third stage.

For the systems gcc-O0, gcc-O2, clang-O0, and clang-O2, we repeat these processes.

It should be noted that the GNU C library, glibc, cannot be generated with anything other than the optimised gcc; as a result, we eliminate all static libraries created during glibc compilation and instead compel the use of dynamic libraries.

B. *Encoding*

Being in possession of the binary files is insufficient for the analysis since they must first be transformed into a vector of features before being sent to the learning network. Since this is the finest granularity that can be achieved with the various compilation settings we might anticipate encountering in a practical instance, we are obviously interested in having a

function-grained analysis. We suggest and contrast two methods: one that calls for exact function limits and disassembly, and the other that relies on a stream of bytes without knowing the contents of the instructions beforehand. In Section IV-A, the efficacy of these two strategies is examined.
Radare22 is used to decode the initial encoding, the one that has to be disassembled, and extract each executable function.

```
            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
```

Figure 1

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 dealing with the x86 64 architecture, we can see a large number of bytes, indicated by the bytes 0x48 in the Figure, that come before each instruction using the rax, rsi, and rsp registers and declare that the registers to be utilised should be 64-bit length. Real functions can have variable lengths, but our networks only accept input vectors of a specific length. Because of this, we must extract the superfluous bytes from the vector in order to retain only the bytes that indicate the operations to be carried out, without any parameters. To reduce space and put more "useful" instructions into the short vector, we do not embed parameters in our representation, unlike earlier studies.

Thus, each moment of time that we encode as an element of our function is actually an opcode of the CPU architecture. For instance, the vector for the first four instructions of the aforementioned picture would be [0x89,0x31,0x85,0x74]. For example, the final two instructions in Figure 1 are constituted of the opcodes 0x85 and 0x0F95 but would have this vector: [0x85, 0x0F, 0x95]. In the event of multibyte instructions, the multiple bytes are interpreted as numerous values. The input vector v's maximum length is set at 2048 bytes a priori. As we anticipate that the most useful actions will occur towards the conclusion of a function rather than its beginning, which comprises initialization, extra data is pre-truncated, and functions that are too short are pre-padded with zeroes, as this has been proved to be better for LSTMs.

From here on, this representation will be referred to as opcode-based.

The second encoding is more simplistic and has already shown promise in earlier studies on learning function boundaries.

To create this second form, the executable's.text section is dumped and divided into fixed-sized pieces using readelf. In order to imitate a real-world situation, we once again use 2048 bytes as the length of v. However, we test the networks' accuracy in identifying the compilation parameters for these chunks while changing their size.

```
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
```

Figure 2

case where functions can have different length.

The fact that we have no idea whether the raw data in this second form represents instructions or stack data is a disadvantage. However, disassembly, which often takes several minutes and is necessary for the initial representation provided and the prior study, is not necessary.

C. *Padding*

The input vector length and function length are the same for the first encoding shown in Section II-B. As a result, the function length is constantly variable, necessitating the previously discussed padding. The second method, on the other hand, does not require padding because it just needs a specific amount of sequential data from the binary.

However, we found that both CNN and LSTM are unable to handle padding during evaluation since they are given chunks that are always a constant size. The inference of a chunk that has been pad dinged with zeroes by more than 60% of its length leads in an undesirable decline below 80% accuracy when training with fixed size chunks, according to experimental data. If we want to infer less data or only a piece of the executable, we would need to train separate models for varied input sizes, which might be harmful in practice.

We truncate the chunk to a random length in the range to solve this issue. Then, using the first strategy outlined in Section II-B, we pre-pad its length. In order for the chunk to still be classifiable, the value 32 was chosen. If we padded too much, we might end up with chunks for which classification is impossible due to a lack of sufficient data.

Figure 2 provides an illustration of this: each line represents an input sequence, with the left block's input sequence coming before padding, and the right block's input sequence coming after padding. The amount of input data that will be shortened is indicated in red. This segment's length is chosen at random within the interval constraints indicated earlier in the section. We can see that the identical amount has the prefix "0" added to it in the right block.
By using these modified chunks to train neural networks, it is possible to predict sequences that include up to 99.5% zeroes with greater than 90% accuracy.

For our analysis we use three different networks modeled as follows: a Fully Connected Dense Network,

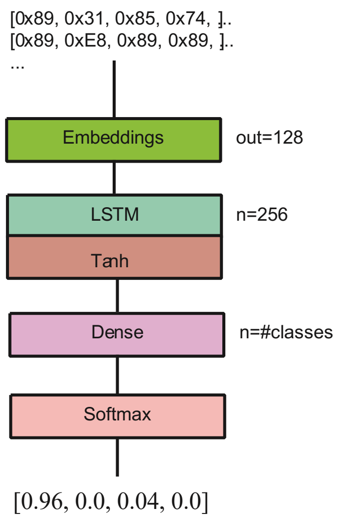


Fig. 3. LSTM model structure

Convolutional Neural Network (CNN) and Long Short Term Memory (LSTM). The success of the latter two networks in image recognition or natural language processing led to their selection. As a result, we are treating our optimization detection problem as a pattern recognition problem. A network may identify a certain optimization as a pattern in the input vector. The original layers structure has undergone minor perturbations in order to find the ideal arrangement for our issue.

Since Dense is just used as a comparison to highlight its inefficiencies, it is not discussed in full here. It is made up of three thick layers with complete connections, each with 2048, 1024, and 512 units. The last layer, which is in charge of making predictions, consists of 1 unit in the binary case and 4 units in the multiclass scenario. ReLU is utilised as the activation function after each layer, followed by a 20% Dropout. The final layer's activation function is a Sigmoid for the binary case and a Softmax with four classes for the multiclass scenario, and the optimizer utilised is Adam with a learning rate of 103.

Figure 3 shows the second model. Given that we packaged our sequence of bytes as a time series and LSTMs' propensity to excel in such tasks, this model implements a straightforward LSTM. In reality, LSTMs feature unique "memory" cells that may be utilized to recall a certain input or pattern even over a lengthy period of time. Our main concept is to train this type of model over a significant number of binary bytes to memorize a certain pattern that represents the compiler or the optimization level.

The model is rather simple, as shown in the image, and is made up first of all of an embedding layer with 256 as the vocabulary size (because we are using bytes in the range 0x0 to 0xFF) and 128 as the dimension for the dense embedding. This layer converts positive integers into a dense, fixed-size vector that the LSTM can comprehend. The real learning is then done using the LSTM layer, which has 256 units. An hyperbolic tangent (tanh) is used as the activation function in this layer. By selecting samples from a uniform distribution, the kernel is initialized $[-64^{-\frac{1}{2}}, 64^{-\frac{1}{2}}]$.. 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 final model is built on a convolutional neural network and was inspired by recent developments in image classification and recognition. The concept is to extract highly dimensional information from a series of raw bytes supplied as input using a collection of convolutions. Figure 4 depicts the structure.

Given that it serves the same purpose in both versions, the initial layer is the same in both. Then, using progressively more filters, three blocks of convolution, convolution, and pooling are utilised. The convolution layer in the figure stands for kernel size 3, number of filters 32, and strides 1. These blocks employ convolutions to extract features from the bytes' order, and pooling to make those characteristics independent of the sequence's location.

Since the ReLU has a vanishing gradient issue, the Leaky ReLU is employed in its stead. In actuality, the leaky variation of the ReLU function returns an in our instance 0.01 in order to maintain the neuron's viability, whereas the ReLU function returns 0 for values less than 0. We utilise the leaky version to prevent the gradient from disappearing and staying at 0 for the remainder of the training since, as was mentioned in Section II-B, we are utilizing heavily padded sequences. A ReLU activation is employed after the standard Dense and Sigmoid for binary classification or Dense and Softmax for multiclass classification, followed by one last fully connected layer made up of 1024 neurons. Also, in this case the optimizer is Adam with learning rate 10^{-3}.

For binary classification and multiclass classification, all models employ categorical cross-entropy as the loss function, respectively .

The Hyperband technique has been used to estimate the reported hyperparameters for the LSTM and CNN [26]. With the exception of kernel size and strides, we searched in the range [32,1024] using the power of two. The set "3,5,7" was used in the space search for the kernel size. Instead, the set 1,2 was used as the space search for the stride value.

III. EVALUATION

After the preprocessing described in Section II-B, we conducted tests using an Nvidia Quadro RTX5000 on the data reported in Section II-A. However, because there were so many binary data points, we ended up with millions of input vectors. Thus, we could securely divide the data into separate sets while maintaining a large number of samples for each group. These sets have a split ratio of 50% for training, 25% for validation, and 25% for testing.

Since no augmentation was done and no duplicate sequences were gathered, every sample used for training, validating, and testing was completely unique. Additionally, duplicate samples (such as system calls) were taken out of each set. No features have been eliminated because they could have been too tiny. Then, each class was balanced by randomly eliminating data from the training set, validation set, and testing set. The number of samples utilized following this preprocessing procedure is shown in Table I.

TABLE I

NUMBER OF SAMPLES FOR EACH CONFIGURATION

Dataset	Training	Validation	Testing
D_{func}	44876	22438	22438
D_{gcc}	46378	23190	23188
D_{clang}	388372	194186	194186
D_{mixed}	92746	46374	46374
D_{multi}	92756	46380	46376

In the Table the dataset names are the following:

Dfunc

Dataset is made up of extracted opcodes that follow instructions. B's It is made up of the gcc-O0 and gcc-O2 programmers that make up the Linux from Scratch build classified in binary form.

Dgcc

Dataset is made up of unprocessed results from the gcc-O0 and gcc-O2 programmers that make up the Linux from Scratch build classified in binary form.

Dclang

Dataset is made up of unprocessed results from the clang-O0 and clang-O2 programmers that make up the Linux From Scratch build. classified in binary form.

Dmixed

Dataset is made up of the unprocessed results of all the Linux From Scratch images that were produced. Both gcc-O0 and clang-O0, as well as gcc-O2 and clang-O2, have been combined into a single class. The O0 and O2 classes in this instance have been balanced to include the various examples from gcc and clang. classified in binary form.

Dmulti

Dataset is made up of programmer output from all configurations in its raw form. Every class is independent, in contrast to Dmixed. Multiple-class classification.

40 epochs of training have been completed with 256 sample batch sizes. An early stopper was used, terminating the learning after three epochs if the loss function on the validation dataset hadn't improved by at least 103. The replication kit in contains more information on the training procedure. The time needed for each sample during training and inference is displayed in Table II.

The following research inquiries are what we hope to address in this section:

• RQfunc: Is it preferable to feed raw bytes or opcodes encoding?

• RQbinary: How do the different models perform when determining the amount of optimization?

TABLE II

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

• RQmixed: How well do the various models identify the optimization level when mixing different compilers?

• RQmulticlass: How do the different models perform when identifying the compiler and optimization level being used?

• RQpad: Does padding during training enhance network performance?

We carry out the assessment for progressively padded values of v in order to respond to these queries. This implies that we test the model's performance using only inputs made up of one byte, two bytes, and so on up to the 2048-byte limit for the input vector. We can gain insight into the model's performance in the event that fewer than 2048 bytes are available in this way.

We give the de-compilation results of three files, a little, medium, and big sized, resulting from the clang-O0 compilation in Table III to better compare the achieved results with the typical scenario.

TABLE III

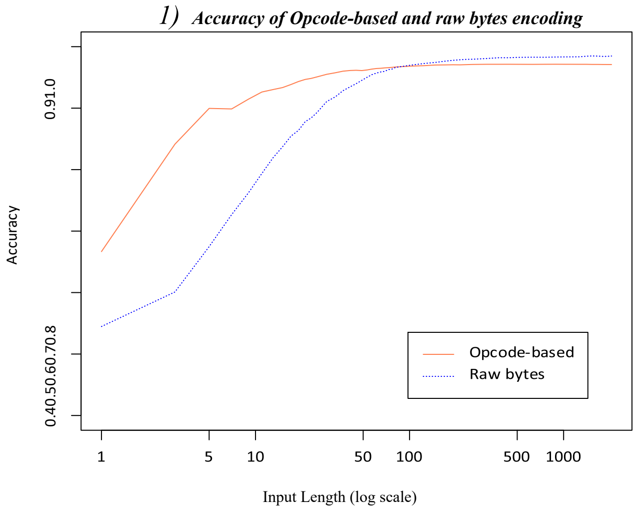
STATISTICS ABOUT A SMALL, MEDIUM AND HUGE FILE, WITH MINIMUM, MAXIMUM AND AVERAGE FUNCTION LENGTH IN BYTES

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. *A. Encoding*

In order to answer RQ_{func}, we train an LSTM over two different datasets: one on D_{func} and one on 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

1) Accuracy of Opcode-based and raw bytes encoding



2) Fig. 5. Accuracy of the opcode encoding compared to raw encoding samples in the testing set with progressively increasing bytes. The plot in Figure 5 shows the results.

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 highvalued 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*

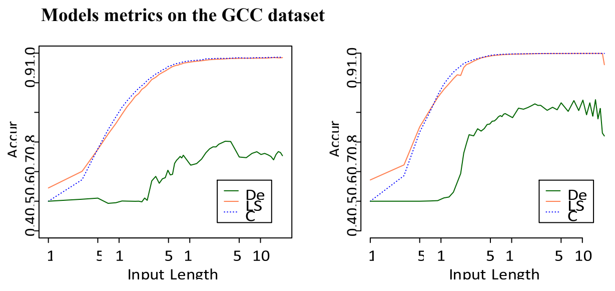


Fig. 6. Accuracy of the various model in the D_{gcc} dataset.

accurate prediction. We can thus answer RQ_{func} as follow:

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

B. *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 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%

IV. 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.

V. 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 reoptimization," 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.