Received November 4, 2021, accepted November 24, 2021, date of publication December 6, 2021, date of current version December 17, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3132950

Identifying Compiler and Optimization Level in Binary Code From Multiple Architectures

DAVIDE PIZZOLOTTO AND KATSURO INOUE, (Member, IEEE)
Graduate School of Information Science and Technology, Osaka University, Osaka 565-0871, Japan

Corresponding author: Davide Pizzolotto (davidepi@ist.osaka-u.ac.jp)

This work was supported by the Japan Society for the Promotion of Science (JSPS) KAKENHI under Grant 18H04094.

ABSTRACT While compiling a native application, different compiler flags or optimization levels can be configured. This choice depends on the different requirements. For example, if the application binary is intended for final release, the flags and optimization settings should be set for execution speed and efficiency. Alternatively, if the application is to be used for debugging purposes, debug flags should be configured accordingly, usually involving minor or no code optimization. However, this information cannot be easily extracted from a compiled binary. Nonetheless, ensuring the same compiler and compilation flags is particularly important when comparing different binary files, to avoid inaccurate or unreliable analyses. Unfortunately, to understand which flags and optimizations have been used, a deep knowledge of the target architecture and the compiler used is required. In this study, we present two deep learning models used to detect both compiler and optimization level in a compiled binary. The optimization levels we study are O0, O1, O2, O3, and Os in the x86_64, AArch64, RISC-V, SPARC, PowerPC, MIPS, and ARM architectures. In addition, for the x86_64 and AArch64 architectures, we also determine whether the compiler is GCC or Clang. We created a dataset of more than 76000 binaries and used it for training. Our experiments showed over 99.95% accuracy in detecting the compiler and between 92% to 98%, depending on the architecture, in detecting the optimization level. Furthermore, we analyzed the change in accuracy when the amount of data was extremely limited. Our study shows that it is possible to accurately detect both compiler flag settings and optimization levels with function-level granularity.

INDEX TERMS Compilers, deep learning, static code analysis, reverse engineering.

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 occurs frequently. During this transformation, the compiler is passed several flags via the "build" commands and settings contained within a Makefile or equivalent. This informs the compiler of the developer's intent to retain or omit some information or to modify the original code in an optimized version.

These flags can be used to optimize faster executions, smaller sizes, and lower energy consumption [1]. However, the flags are not explicitly recorded in the binary itself, as they are completely unnecessary for the machine to execute the binary code.

Moreover, the compiler itself is not easily identifiable. There is no standard way to record this information, and although some compilers write a comment in the binary itself, it is easily skipped or duplicated and not guaranteed to be parsable. For example, if a file compiled with the Clang compiler is linked with a library compiled with GNU compiler collection (GCC), this comment will contain both signatures.

However, this information is extremely valuable for various applications, such as categorizing an older build, finding vulnerabilities [2], finding similarities in binaries [3], rewriting a binary [4], or providing accurate bug reports in case the compilation environment cannot be controlled [5]. A simple example of the latter case could be a library that has incompatibilities only with a specific compiler, in a product published by a different vendor than the library developer. Pallister *et al.* have shown that different optimization options can have a significant impact on final energy consumption [6]. The use of precompiled libraries could be problematic in embedded applications where energy consumption is a concern.

Knowing the compilation flags could even be helpful when performing binary analysis. In their work,

The associate editor coordinating the review of this manuscript and approving it for publication was Shen Yin.

Pewny *et al.* reported that applying their analysis to different compilation flags significantly affects accuracy [7]. In this case, our work will help to provide confidence in the analysis results, because the differences between the optimization levels of O2 and O3 are much less pronounced than those of O0 and O3.

Although there are several papers on detecting the compiler [5] and toolchain [8] used, these methods do not rely on automatic learning approaches. Therefore, a significant effort is required to detect the above information in different architectures. The new architecture must be studied and understood to check if and how the information can be retrieved. In contrast, with a machine learning-based approach, it is sufficient to provide new data and re-run the training to detect a new compiler or flag. With the automated dataset generation provided in our work, the time required to generate this data is a couple of hours for each optimization level that we want to classify.

In this study, we present our approach for recognizing both compiler and optimization levels using a long-short term memory network (LSTM) [9] and a convolutional neural network (CNN) [10] within different architectures. We analyzed common optimization levels on Linux binaries, compiled with either GCC or Clang in two different architectures, and with GCC only in seven different architectures. We want to identify optimization levels ranging from non-optimized code (O0), code that is optimized for speed with different levels of aggressiveness (O1, O2, O3) or code optimized for small binary size (Os). Although we are not the first to tackle this problem [11], the novelty of our research can be summarized as follows:

- The creation of a huge dataset with automated replication scripts consisting of 76630 compiled files. These files come from seven different CPU architectures and two different compilers, with a combined total of 123 GB of stripped binary data.
- The implementation and tuning of a neural network structure that outperforms existing work in flag detection.
- An analysis that examines the minimum possible number of raw bytes that are required to obtain accurate predictions.

This study is an extended version of our previous work [12]. The main differences between our previous study and the current study are as follows.

- The number of optimization levels test was increased from {O0, O2} to {O0, O1, O2, O3, Os}.
- The number of architectures tested have increased from {`x86_64`} to {`x86_64`, `AArch64`, `RISC-V`, `SPARC`, `PowerPC`, `MIPS`, `ARM32`}. These additional architectures require a completely different cross-compilation approach for generating the dataset, as described in Section IV-A. We also made considerable efforts to automate the generation of the dataset. In our previous work, compilation was a manual task, while now we provide several scripts to automate the process. The main advantage of this automated approach is the ability to generate a different dataset with different compilation flags without the user having to do anything. Although this does not work for additional architectures, it still provides an effortless way to generate the dataset with additional flags.
- The evaluation has been completely reworked, considering the expanded category set. Moreover, in this extended version, we tested the feasibility of our study in a scenario with function-level granularity. We also examined the distribution of flags in the Ubuntu and MacOS operating systems.

The rest of our paper proceeds as follows. Section II presents the motivation for our work. Section III discusses related work in the field of binary file analysis with machine learning. Section IV presents the problem and our approach and Section V presents the empirical evaluation. Section VI compares our choices with another similar work in the field and discusses them in terms of the results obtained. Section VII describes the limitations of our study and Section VIII concludes the paper. In addition, Section IX provides instructions to download our dataset and source code.

II. MOTIVATION

After briefly introducing the motivation behind our work in Section I, we provide an example of the reasons behind our work in this section. Although there are several reverse engineering tools such as IDA[1] or Ghidra,[2] these usually do not detect the flags used during compilation or at link time. This is because a user is interested in extracting and collecting information from single binary files as part of reverse engineering. Despite this, in some applications knowing the original binary flags is almost mandatory. For example, Wang *et al.* showed how in binary rewriting most of the failures are due to compiler optimizations [4].

The decompilation scenario is also equally interesting. According to Katz *et al.* [13], the presence of optimizations reduces the success rate in correctly decompiling an executable. Moreover, their results are based on recurrent neural networks (RNNs), so we expect traditional approaches with hand-crafted rules to be more susceptible to underlying optimizations. With a higher level of optimization, we expect the decompiler to output a higher amount of `goto` statements. If this is the case, knowing the optimization flags may give an indication of the expected accuracy of a decompiler, knowing that a heavily optimized code may decompile into source code that is more difficult to analyze.

Nevertheless, the focus of our study was on code analysis. When comparing multiple binaries, knowledge of the compilation flags becomes much more useful, almost mandatory. As reported in Section I, Pewny *et al.* in their analysis of cross-architectural binary code devoted an entire research

[1]https://hex-rays.com/
[2]https://ghidra-sre.org/

question investigating the presence of false/true positives at different optimization levels [7]. Ultimately, they concluded that comparing non-optimized to optimized code results in significantly lower accuracy. In our internal study, we came to a similar conclusion. Comparing control flow graphs (CFGs) of functions, even within the same architecture but with different compilers or optimization flags, is dominated by false positives. The impact is so great that the comparison is usually impractical when dealing with different compilers or O0 versus O2, where it is perfectly fine when comparing O1 with O2 or O3. Thus, performing analyses or comparisons between binaries may lead to results that are either good or completely unusable, depending only on the compilation flags used. These two contrasting results have led us to look for a way to detect in advance the presence of optimizations. This is a way to determine whether a comparison or analysis between binaries is feasible, and if the results will be reliable.

In our previous study, we focused only on the presence/absence of optimizations, the most influential factor in determining a poor comparison between two binaries. In this extended version, we also tried to detect the degree of optimization. In fact, we can expect a lesser loss of accuracy when comparing, for example, an O1 optimized code to an O3 optimized code.

III. PREVIOUS WORKS

Binary file analysis is widely used in the field of security. Recently, machine learning techniques have been used to aid malware detection. Pascanu *et al.* used recurrent neural networks (RNNs) [14] to extract malicious features from a binary file in an unsupervised manner, which was extended with convolutional neural networks by Athiwaratkun *et al.* [15].

Related works dealing with compiler flags, instead, focuses on the effects of these flags rather than their detection. Work performed by Triantafyllis *et al.* focused on exploring optimal compiler flags [16] as did the work of Hoste *et al.* [1]. In recent years, several machine learning-based techniques have been developed [17], [18]. Older techniques focus on the use of machine learning to reduce the number of iterative compilations required to obtain a good set of flags [19] and to help approximate NP-hard problem efficiently, like phase ordering [20]. Instead, more recent techniques use deep learning to detect function boundaries, a work by Bao *et al.* [21], which was then extended by Shin *et al.* [22]. Chua *et al.* instead detected function types using RNNs [23] whereas He *et al.* attempted to recover the debug symbols from a stripped binary [24].

To the best of our knowledge, the only work attempting to detect flags in an existing binary, rather than optimizing them, is that of Chen *et al.* [11]. The main differences between their study and our work are as follows:

- We investigate the detection of not only flags but also compilers.
- We investigate the detection in seven different architectures instead of only one.
- Our analysis aims not only to maximize the accuracy but also to minimize the required input.
- Our dataset is more than 100 times larger [25], disproving some claims of previous research.

IV. APPROACH

The problem we are trying to solve is to identify the optimization level and possibly the original compiler used to compile from source code to binary code, when only a portion of the binary code is available. Specifically, given a sequence of bytes v of arbitrary length coming from a binary, we want to train a classifying function $\mathcal{M}_{flags}$ capable of predicting the compilation flags and a classifying function $\mathcal{M}_{compiler}$ predicting the compiler used.

With $\mathcal{M}_{flags}$, we try to classify the optimization level used in the input binary. In our study, we target the commonly used optimization levels $\{O0, O1, O2, O3, Os\}$.[3] We trained different $\mathcal{M}_{flags}$ for each of the seven architectures we studied, namely `x86_64`, `AArch64`, `ARM32`, `MIPS`, `PowerPC`, `RISC-V`, and `SPARC`, and expect a user to select the prediction model according to the input architecture. The architecture of a binary is easily recognizable by tools such as `file`, so this fact is not a limitation and simplifies training.

Similarly, with $\mathcal{M}_{compiler}$ we classify the compiler between `gcc` and `clang`, which means that the compiler analysis is a binary classification. This is done only for the natively generated architectures, such as `x86_64` and `AArch64`, for reasons explained in Section IV-A2. Thus, we have trained two different $\mathcal{M}_{compiler}$.

Our goal is not only to maximize accuracy, but also to keep the sequence of bytes v as small as possible; as such, we dedicate Section Section IV-B to explain how the binary code is transformed into v (or several vs), the input expected by our learning network. To compare the performance of different models, we trained all the aforementioned configurations using a feed-forward Convolutional Neural Network $\mathcal{M}^{CNN}$ and a Long-Short Term Memory Network $\mathcal{M}^{LSTM}$, producing in total $7\mathcal{M}^{CNN}_{flags}$, 7 $\mathcal{M}^{LSTM}_{flags}$, 2 $\mathcal{M}^{CNN}_{compiler}$ and 2 $\mathcal{M}^{LSTM}_{compiler}$. These networks are trained in several different datasets, explained in detail in Section IV-A, and their prediction results are compared.

More details about the network models can be found in Section IV-D.

A. DATASET

To train our networks, we must first collect the data. Our networks perform supervised learning, so it is necessary to partition the binary code by optimization level and compiler used.

Although this task may seem trivial, as there are plenty of open-source software programs that can be compiled with the desired flags, several precautions are required during the linking phase. This is because although we can choose both

[3]Note that some applications might use additional "hand-picked" flags. This limitation is discussed in Section VII

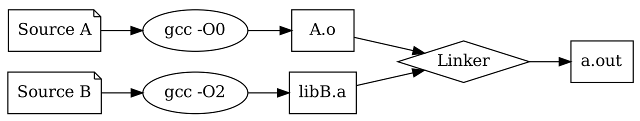


FIGURE 1. Linking an executable may include binary data compiled with different flags.

the optimization level and compilers, we have no guarantees about the environment that performs the compilation. There are several libraries available to be statically linked, and we know nothing about the compilation settings used for these libraries. This problem is highlighted in Figure 1.

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 while generating the dataset. In Figure 1, this is shown in the *Source B* path, where, a source originally compiled with an optimization level of O2 is linked within the same binary with an optimization level of O0. This implies that these libraries could irreversibly contaminate our build, because we lack information about their creation.

Possible options for solving this problem would be:

1) Use a dataset composed of object files prior to linking.
2) Edit the build script for the generated binary to exclude static linking.
3) Create a system with only shared libraries. Then use this system to build the dataset.

In our study, we chose options 3 for the following reasons: Option number 1 would not provide a realistic case study, as several optimizations can also be performed at link time [26]. Option 2, could be "too risky" as it involves manually checking various compilation 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 hard-coded parameters, others use environment variables, and others recursively integrate files. Checking, understanding, and modifying all these build scripts correctly is entirely possible but carries a high risk of error.

Option 3 can guarantee the absence of contamination, but requires some care, as one needs to isolate from the existing environment. While generating our dataset, in order to ensure this constraint was respected, we used two different approaches depending on whether the build architecture was matching the target one or not. The final dataset is publicly available on Zenodo at the following link [25].

1) DATASET: NATIVE COMPILATION

The dataset we use for the native compilation contains all software listed in the *Linux from Scratch* book,[4] version 9.1-systemd, published on March 1, 2020. In addition, we added to every dataset the LLVM suite version 10.0.0, which is required for building Clang.[5] To solve the static linking problem within a native build system we performed the following steps:

1) We built a toolchain with no particular compiler and optimization level from the host machine. We ensured that only shared libraries were built in this toolchain.
2) We created a chrooted environment containing only the toolchain to isolate it from the original build system.
3) We built the actual dataset, with the desired compiler and optimization level.

After building all software binaries, we then strip and use each ELF file.

2) DATASET: CROSS COMPILATION

The cross-compilation approach we use is based on the presence of readily available toolchains in the Ubuntu Package repository.[6] For the following study, we need the *gcc*, *g++* and *binutils* packages for each architecture we want to target. For cross-compiled architectures, we did not build a Clang dataset and limit our analysis to GCC only.

Unlike in the native compilation presented in Subsection IV-A1, the linker is not capable of linking binaries from the build system because they are compiled for a different architecture. However, existing toolchains may still ship static libraries, requiring some care to ensure either these libraries are built with the correct flags or not present at all. Aside from that, chrooting is not neeeded, meaning that we can fully automate the entire building process, which normally ends with the execution of the chroot command. Pointers to the script used to perform this automated building can be found in Section IX.

However, not all software used in the native building supports cross-compiling. Despite heavily editing most scripts, some software, such as Perl, fails to cross-compile. For this reason, when cross-compiling, we use a slightly different dataset and limit our study to the GCC compiler. This new dataset comprises all the software from LFS that supports cross-compilation as well as some software coming from *Beyond Linux from Scratch*.[7] Due to space limitations, we cannot list every piece of software used in this building process. For a comprehensive list see the `resources/scripts` folder in the repository listed in Section IX.

As with native compilation, each ELF file generated is stripped and used in the dataset.

B. PREPROCESSING

In this section, we explain the additional preprocessing before the input vector is fed into the networks. In the evaluation, we prove that advanced encoding is unnecessary, but we report it here anyway because previous research has used it.

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

[5]https://releases.llvm.org/10.0.0/

[6]https://packages.ubuntu.com

[7]http://www.linuxfromscratch.org/blfs/index.html

```
4889442418   mov qword [rsp+0x18], rax
31C0         xor eax, eax
4885FF       test rdi, rdi
7423         je 0x25
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 0xE5F
4885C0       test rax, rax
0F95C0       setne al
```

FIGURE 2. **Portion of a disassembled function.**

The goal of this preprocessing is to transform a binary file, usually composed of a sequence of data and instructions, into one or more input vectors for the automated learning step. We are naturally interested in correctly classifying the smallest possible input, with the finest grain being the function grain. We compare two approaches: one using a stream of bytes without prior knowledge of the underlying data and one that requires disassembly and precise function boundaries. The effectiveness of these two approaches is analyzed in Section V-C.

The first approach, referred to as "raw bytes" throughout the paper, has been shown to be effective for learning functions boundaries in previous research [21].

To generate this representation, we use `readelf` to dump the `.text` section of the executable and divide it into fixed-size chunks. We chose a-priori 2048 bytes as the maximum size of this chunk of bytes v. However, we also evaluated the precision of the networks in detecting the compilation settings for these chunks as their size varied, in order to simulate a case where we want to detect the optimization of a single function.

One drawback of this representation is that we do not know whether the raw data represents instructions or stack data. In contrast, if we want to classify an entire executable, disassembly is not required, a step that usually requires several minutes. Even when we want to classify at function granularity, based on the research of Bao *et al.* [21], we can extract function boundaries and perform classification without disassembling the executable, which is not only slow, but is also inherently difficult and prone to error in some architectures [27].

In the second representation, the one that requires disassembly, *radare2*,[8] is used to extract each function from the executable. The results are shown in Figure 2.

In the figure, the left column represents the raw bytes written in the binary and the right column their translations in Intel Assembly syntax. The example is taken from `x86_64` disassembled code. We can see many bytes specifying that the registers to be used should have a length of 64 bits, represented by the red bytes in figure, preceding every instruction involving `rax`, `rsi`, `rsp` registers. This is a problem, because real functions can be of arbitrary length, but our networks support fixed-length vectors as input. We expect these extra bytes to contribute almost nothing to the final result and thus decide to remove them and keep only the byte(s) representing the operations to be performed, without parameters. Unlike the previous research, we also remove the operands in our representation, in order to save more space and accomodate even more "valuable" instructions inside the limited length vector [11]. Finally, only the blue bytes in Figure 2 were encoded in our representation. Extra data are pre-truncated, because we expect the most useful operations are at the end of a function and not at the beginning, which contains the initialization. Insufficiently long functions are pre-padded with zeroes, as pre-padding has been proved to be better for LSTMs [28]. From now on, we will refer to this representation as "encoded".

In both representations, we feed our data to the networks as a time series, where each point in time is actually a byte of data from the binary file. For example, the first two instructions of Figure 2 would have this vector in the raw byte approach: `[0x48, 0x89, 0x44, 0x24, 0x18, 0x31, 0xC0]`. In the encoded representation instead they would be `[0x89, 0x31]`.

C. PADDING

For the encoded representation presented in Section IV-B, the input vector length is equal to the function length. This makes it necessary to pad the data, as the function length is always different. However, the raw data approach, requires a fixed amount of sequential data from the binary. As any data in any part of the `.text` binary section can be used as an input, in principle, no padding is strictly required.

However, we determined that by always providing unpadded vectors, both CNN and LSTM are unable to deal with padded data during evaluation. The experimental data in Section V-D shows that when training with unpadded inputs, the inference of a vector padded with zeroes by more than 60% of its length results in a 10% accuracy drop. This can be detrimental in a real case, as it would be necessary to train different models for different input sizes if we want to infer a smaller amount of data or just a portion of the executable.

To solve this problem, we truncate a random number of bytes in the interval $[0, \alpha]$ where

$$\alpha = len(v) - 32$$

The value 32 has been chosen so that the input chunk v to be still classifiable: if we pad too much, we might get chunks where the classification is impossible due to lack of enough information. This number is also extremely conservative: consider that in `x86_64`, for example, just calling

[8]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
```

FIGURE 3. Truncation of input sequences on the left and subsequent padding on the right. The amount of truncated bytes in this Figure is symbolic.

an imported function requires at least 11 bytes of data, and translates to a single opcode.

The random amount is defined by an exponential distribution. Our intention is to use a distribution where 99% of the values fall within the above interval, while clamping the outliers to 32. In this case, the network would predominately receive low-padded vectors, while occasionally encountering a mostly-padded vector. With the exponential cumulative distribution function as $y = 1 - e^{-\lambda x}$ we fix y to 0.99 and x to α to obtain the λ shown in Equation 1.

$$\lambda = \frac{2 \ln 10}{\alpha} \tag{1}$$

We then use this λ in the exponential distribution generating the random number of bytes that should be truncated for each input. After truncating the input, we prepad it by adding zeroes.

An example of this can be seen in Figure 3, where each line represents an input sequence before padding on the left block and after padding on the right block. The red part represents the amount of input data that will be truncated. The length of this part is decided randomly within the interval bounds previously mentioned. On the right block we can see that the same amount is replaced by prepending zeroes.

Evaluation of this padding is provided in Section V-D.

D. NETWORKS

For our analysis, we used two different networks: a LSTM [9] and a feed-forward CNN [10]. These networks have been chosen due to their successful applications in natural language processing or image recognition. In fact, we model our optimization recognition problem as a pattern recognition problem: a particular optimization can be recognized by a network as a pattern of opcodes in the input sequence of bytes.

The first model is shown in Figure 4. This model depicts a simple LSTM, given that we encoded our sequence of bytes as a time series and the ability of LSTMs to perform well on this type of problem. LSTMs, in fact, have special ‘‘memory’’ cells, that allows them to memorize a particular input or pattern even in long sequences [9]. Our core idea is to train 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.

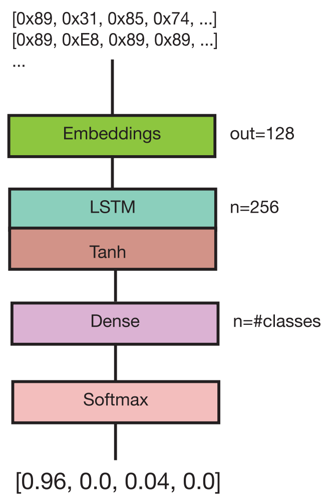


FIGURE 4. LSTM model structure.

As we can see from the figure the model is pretty straightforward. It is composed of an embedding layer with 256 as vocabulary size (because we use bytes in range 0 × 0 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 a hyperbolic tangent (tanh) as the activation function. The kernel is initialized by drawing samples from a uniform distribution in $[-64^{-\frac{1}{2}}, 64^{-\frac{1}{2}}]$. The last part of the network is a dense layer with 1 output and Sigmoid activation for the binary case, dense layer with 5 outputs and Softmax activation for the multiclass case. The optimizer is Adam [29] with learning rate of 10^{-3}.

The second model, is based on the trend in image recognition and categorization [30] and is based on a convolutional neural network. The idea is that a series of convolutions is used to extract highly dimensional information from the sequence of raw bytes passed as input. The Structure is shown in Figure 5.

The first layer is identical to that of the LSTM version, because its utility is the same. Then, three blocks of convolution, convolution, and pooling, with increasing number of filters, were used. In the Figure, the label k3n32s1 for a convolutional layer indicates a kernel size of 3, a number of filters of 32, and a stride of 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 [31] is used in place of the ReLU [32], because the latter suffers from the vanishing gradient problem. Although the ReLU function returns 0 for values less than 0, the leaky variant returns an ϵ, in our case 0.01, to keep the neuron alive. Before output, the final fully connected layer composed of 1024 neurons is used, followed by a ReLU activation and the canonical dense and sigmoid for binary

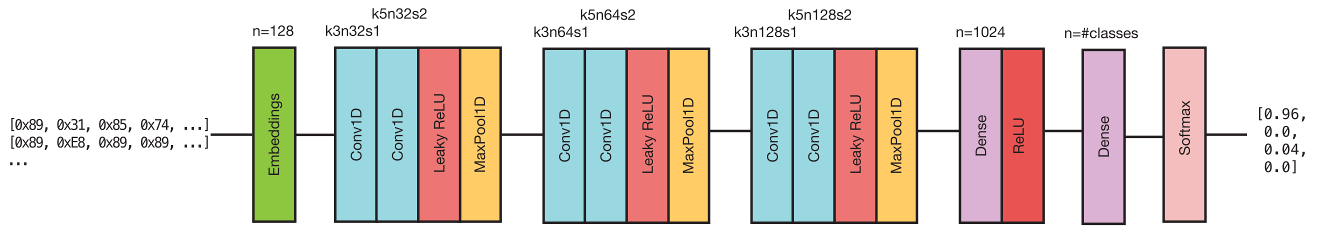


FIGURE 5. CNN model structure.

classification or dense and softmax for multiclass classification. Also in this case the optimizer is Adam with learning rate of 10^{-3}.

All the models use binary cross-entropy as loss function for the binary classification and categorical cross-entropy for the multiclass classification [33].

The presented hyperparameters of both the LSTM and the CNN were estimated using the Hyperband algorithm [34]. We used powers 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 $\{3, 5, 7\}$. Instead, for the stride, the space search was $\{1, 2\}$.

V. EVALUATION

We performed experiments using an Nvidia Quadro RTX5000 on the data presented in Section IV-A after the preprocessing explained in Section IV-B. However, given the high amount of binary data, we obtained a number of input vectors in the order of millions. Thus, we could safely split the data into disjointed sets, while maintaining a high number of samples for each set. Thus, we set training, validation, and testing with split ratios of 50%, 25% and 25% respectively.

No augmentation was performed and no overlapping sequences were collected, with each sample being absolutely unique between training validation and testing. Duplicate samples were removed from each set. These are common especially in the opcode-encoded datasets. Training was performed for 40 epochs using a batch size of 512 samples. An early stopper was employed, which stopped the learning after three epochs if the loss function in the validation dataset did not improve by at least a factor 10^{-3}. In total, we trained 36 models in approximately 550h.

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

- **RQ**$_{accuracy}$: Is it better to use a CNN or a LSTM? What results can be expected from each network?
- **RQ**$_{min}$: What is the minimum number of bytes required to have accurate predictions?
- **RQ**$_{encoding}$: Does extracting data with a disassembler increase the accuracy of the predictions?
- **RQ**$_{pad}$: Does padding during training improve the performance of the networks?
- **RQ**$_{occurrence}$: What are the most common optimization levels in real-case distributions?

TABLE 1. Number of training and testing samples for each architecture. Each sample is composed of 2048 bytes.

Dataset	Training	Testing
$\mathcal{D}_{x86\_64}$	$2.4 \cdot 10^6$	$1.2 \cdot 10^6$
$\mathcal{D}_{aarch64}$	$2.3 \cdot 10^6$	$1.1 \cdot 10^6$
$\mathcal{D}_{riscv64}$	$1.3 \cdot 10^6$	$7.0 \cdot 10^5$
$\mathcal{D}_{sparc64}$	$1.9 \cdot 10^6$	$9.8 \cdot 10^5$
$\mathcal{D}_{powerpc}$	$2.0 \cdot 10^6$	$1.1 \cdot 10^6$
$\mathcal{D}_{mips}$	$1.6 \cdot 10^6$	$8.2 \cdot 10^5$
$\mathcal{D}_{arm}$	$1.2 \cdot 10^6$	$6.2 \cdot 10^5$

RQ$_{accuracy}$ aims to investigate the advantages and disadvantages of using one network over the other for both compiler detection and optimization level detection. This question is then expanded in RQ$_{min}$ to investigate how much the input size can be reduced while still maintaining a sufficiently high accuracy. RQ$_{encoding}$ was investigated to explain our choice of training with raw data. In particular, this contradicts the claim of previous work conducted by us and Chen *et al.* [11]. RQ$_{pad}$, instead, serves the purpose of justifying why in Section IV-C, we claim that padding is necessary while training with raw data. Finally, RQ$_{occurrence}$ concludes our study by running our models in real-case code to obtain statistics about the most used flags, proving the assumption of O2 being the most popular flag as not always correct.

A. ACCURACY

To evaluate the accuracy of both the CNN and LSTM we divided our dataset by architecture. The number of samples we used for each architecture is listed in Table 1. It should be noted that in the worst case we trained with at least 10^6 samples and tested on at least $6 \cdot 10^5$ samples.

The number of features for each sample are 2048 sequential bytes from the specified architecture, categorized by optimization level. In addition, `x86_64` and `AArch64` contains samples compiled with both GCC and Clang. We trained a CNN and a LSTM model for each dataset and obtained the results shown in Table 2 regarding the optimization level detection. Note that all results were obtained with raw encoding and padded data unless otherwise stated.

This table represents the categorical accuracy achieved while performing supervised evaluation of our trained models in the testing data. Being the accuracy a metric defined for binary classification, every time we use this term in a multiclass context, we refer to the following formula in Equation 2,

TABLE 2. Accuracy for each architecture while detecting the optimization level.

Architecture	CNN Accuracy	LSTM Accuracy
`x86_64`	0.8781	0.9291
`AArch64`	0.9181	0.9687
`RISC-V`	0.8427	0.9209
`SPARC`	0.9364	0.9682
`PowerPC`	0.8702	0.9227
`MIPS`	0.9596	0.9837
`ARM`	0.9380	0.9588

TABLE 3. Training time, in minutes, required for each network and architecture.

Architecture	CNN Time	LSTM Time
`x86_64`	361 min	1845 min
`AArch64`	298 min	1764 min
`RISC-V`	220 min	1034 min
`SPARC`	346 min	1397 min
`PowerPC`	398 min	1379 min
`MIPS`	257 min	1362 min
`ARM`	407 min	629 min

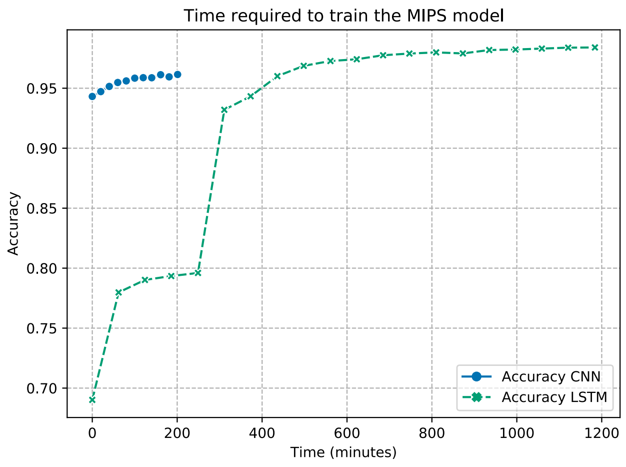


FIGURE 6. Accuracy obtained in the validation dataset at the end of each training epoch.

where tp, tn, fp, fn are true positives, true negatives, false positives, false negatives and k is the number of classes.

$$accuracy = \frac{\sum_{i}^{k} \frac{tp_i+tn_i}{tp_i+tn_i+fp_i+fn_i}}{k} \tag{2}$$

We can note how that the LSTM is always better than the CNN, with the worst accuracy being recorded for `x86_64`, `RISC-V`, and `PowerPC` in both networks. However, the downside of using the LSTM is its extensive training time. We can see this in Figure 6.

The figure shows the times obtained from the MIPS dataset, one of the datasets with the fastest training times owing to its size. It takes approximately seven epochs for the LSTM to reach the same accuracy as the CNN at the end of the first epoch. In addition, the CNN can complete 10 epochs in the time the LSTM is able to complete only 3. We measured similar speeds also during inference, with a CNN two to three times faster than the LSTM. This is not limited to the single architecture we presented in Figure 6. As we can see in Table 3, it applies to any architecture, with the worst case being `x86_64` having an LSTM requiring more than five times the corresponding CNN training time. Note, however, that these times were collected only once. As such, variations, even significant ones, are expected.

To further investigate the accuracy, Figure 7 shows all the confusion matrices for each model trained with the CNN.

The problematic part, as seen from the Figure, is the distinction between O2 and O3. In fact, O0 and O1 can be detected with 99% accuracy in any architecture and Os is never below 96%. O3, however, in the worst case has more wrong classifications than correct ones, as we can see in `PowerPC` and `RISC-V`.

This situation is slightly better when an LSTM is used. The results are shown in Figure 8.

In this Figure, we can see how the LSTM achieves high accuracy in some architectures, namely `AArch64`, `SPARC`, `MIPS`, and `ARM`. The architectures problematic for the CNN remain problematic also for the LSTM, but to a much lesser extent. In fact, no optimization level reports more wrong classification than correct ones, and the worst case is a 70% accuracy for `PowerPC` O2.

To mitigate this problem, we trained two additional datasets: $\mathcal{D}_{merged}$ and $\mathcal{D}_{splitted}$: the first containing all optimization flags but with O2 and O3 merged together, the second containing only O2 and O3.

Figure 9 reports this split dataset situation. We can notice how the CNN network performs slightly better after separating O2 and O3 from the rest of the dataset, whereas the LSTM performs slightly worse compared to the results without separation.

Concerning the compiler detection, results are reported in Table 4.

The table shows how both networks in both architectures perform excellently. Even in the `x86_64` with CNN case, that performed quite poorly in the optimization detection, the incorrect classifications were 587 compared to 1225822 correct classifications. Given the high accuracy of the CNN and its faster speed compared to an LSTM, it should be the preferred choice for compiler detection.

We can thus answer RQ$_{accuracy}$ as follows:

While detecting the optimization level, the LSTM can offer higher accuracy at the price of slower train and inference. The accuracy range from a minimum of 92% to a maximum of 98% depending on the architecture. While detecting the compiler, however, both networks perform well. In this case the CNN is the preferred choice due to its speed advantage and an accuracy of 99.95%.

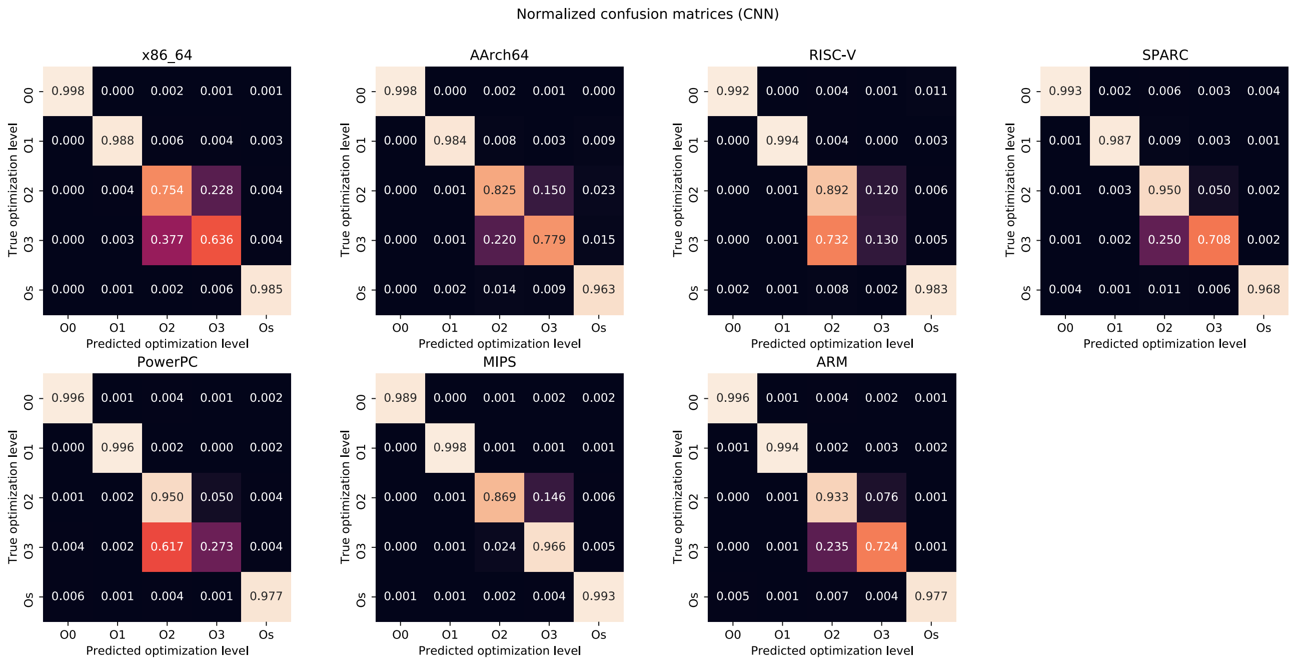


FIGURE 7. **Confusion matrices while detecting optimization level for each architecture. Results obtained with the CNN.**

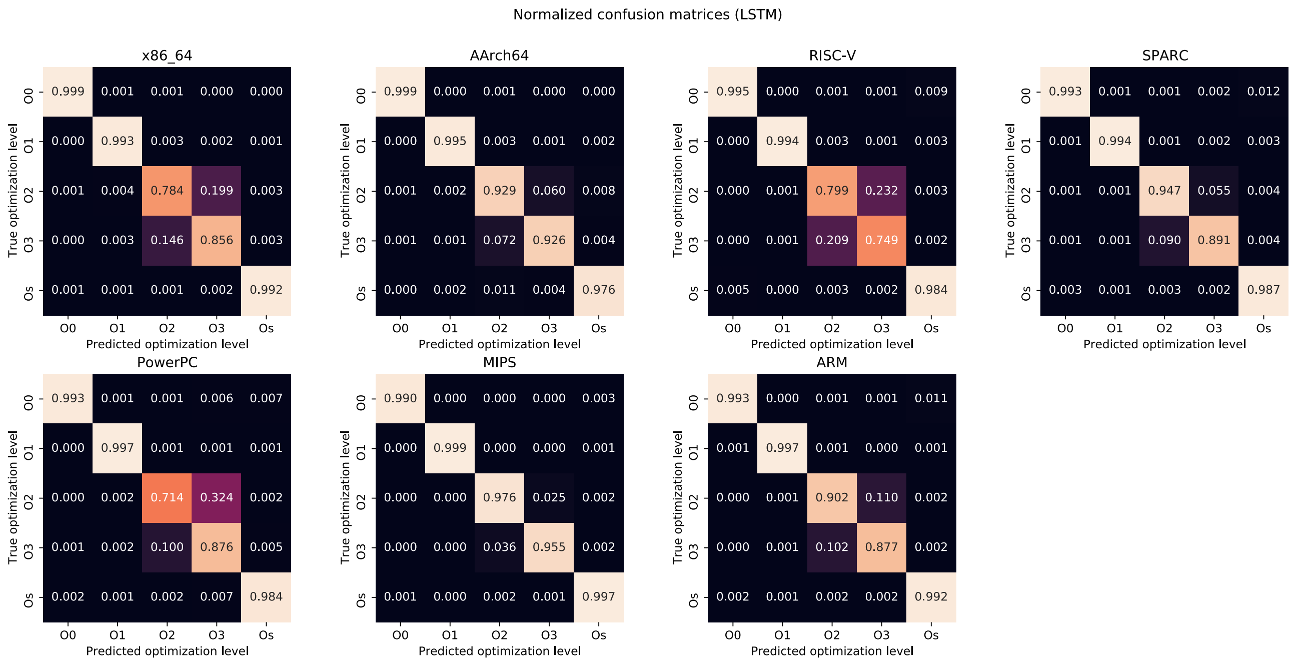


FIGURE 8. **Confusion matrices while detecting optimization level for each architecture. Results obtained with the LSTM.**

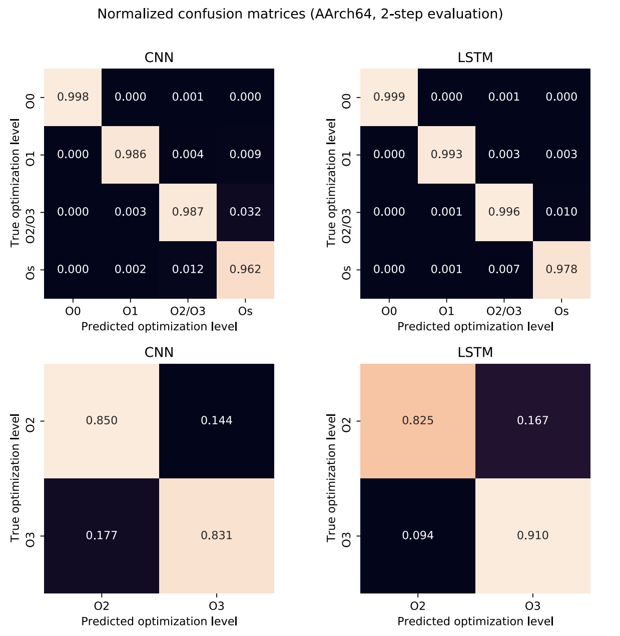
TABLE 4. **Accuracy for each architecture while detecting the compiler.**

Architecture	CNN Acc.	LSTM Acc.
x86_64	0.9995	0.9932
AArch64	0.9996	0.9996

B. MINIMUM BYTES

This section investigates the possibility of detecting the optimization level and compiler with function granularity. To this end, we performed our evaluation for each model while feeding a progressively increasing number of bytes. We thus performed the initial evaluation with only 1 byte for each sample; then, we performed a second evaluation with 2 bytes and so on until we used the full vector length of 2048 bytes.

Figure 10 shows the results obtained using the CNN network. Figure 11, shows the same results but with the LSTM network.



CNN (4-class):

True \ Predicted	O0	O1	O2/O3	Os
O0	0.998	0.000	0.001	0.000
O1	0.000	0.986	0.004	0.009
O2/O3	0.000	0.003	0.987	0.032
Os	0.000	0.002	0.012	0.962

LSTM (4-class):

True \ Predicted	O0	O1	O2/O3	Os
O0	0.999	0.000	0.001	0.000
O1	0.000	0.993	0.003	0.003
O2/O3	0.000	0.001	0.996	0.010
Os	0.000	0.001	0.007	0.978

CNN (O2/O3):

True \ Predicted	O2	O3
O2	0.850	0.144
O3	0.177	0.831

LSTM (O2/O3):

True \ Predicted	O2	O3
O2	0.825	0.167
O3	0.094	0.910

FIGURE 9. Confusion matrices whith split dataset. CNN on left, LSTM on right.

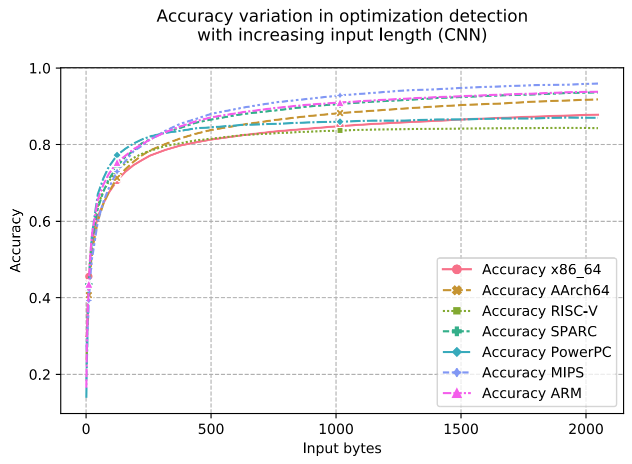


FIGURE 10. Accuracy for the CNN in the optimization detection.

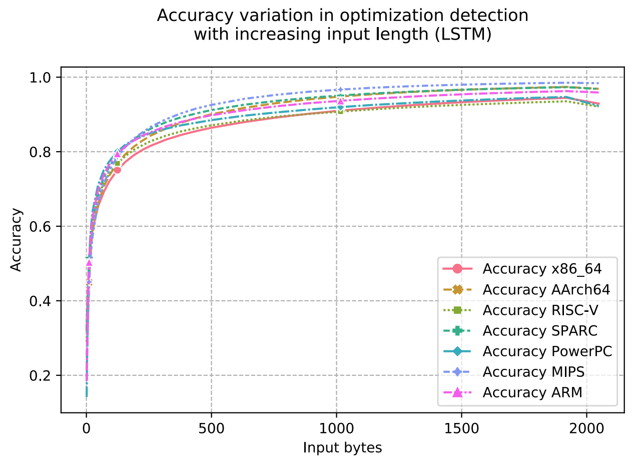


FIGURE 11. Accuracy for the LSTM in the optimization detection.

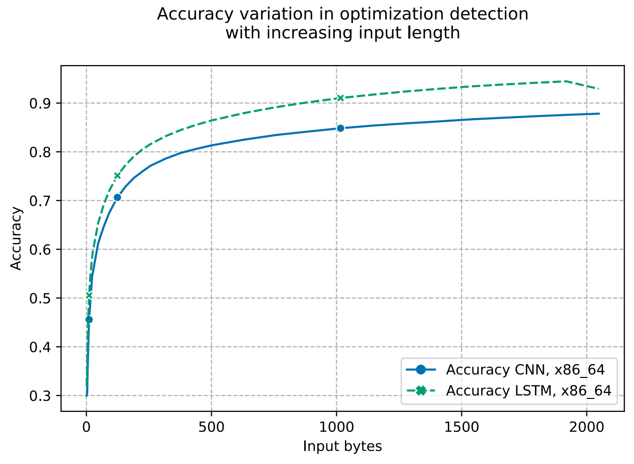


FIGURE 12. Comparison between CNN and LSTM in the `x86_64` optimization detection.

Note that every architecture follows the same detection trend in the LSTM network. In the CNN one, however, the accuracy for `x86_64`, `PowerPC`, and `RISC-V` stops increasing, unlike in the other architectures. These three architectures are the same architecture we classified as "problematic" in Section V-A. In addition, this does not happen for the LSTM network. We can assume the CNN failed to learn how to properly recognize O2 and O3 in these architectures, given the strong similarity between these two optimization levels. As such, additional bytes do not help the network at all, in contrast to the LSTM case.

Additionally, we can note how, with any number of bytes, the LSTM performs definitely better than the CNN.

To highlight this, Figure 12 shows a direct comparison between LSTM and CNN in a single architecture, `x86_64`. This figure, shows how there is always approximately 5% more accuracy in the predictions of an LSTM compared to the predictions of a CNN.

Having analyzed how the overall accuracy varies when the input length changes, we now want to check whether the average function length is sufficient to achieve good accuracy. To do this, however, we need to gather statistical data on binary files. The idea is to calculate the average function length at each optimization level and check the accuracy of the network for that particular level at that particular length.

We disassembled every binary for every optimization level and compiler and counted the number of bytes that compose each function. The results are shown in Figure 13.

The reported number is the median, calculated over $47 \cdot 10^6$ functions across all architectures. We chose to show this average metric, as opposed to the mean, as we want to draw conclusions based on the "typical" function length. This is also a much more conservative approach, given that the mean is influenced by some outliers with a very high number of bytes, on the order of 10^6. Although Figure 13 shows the

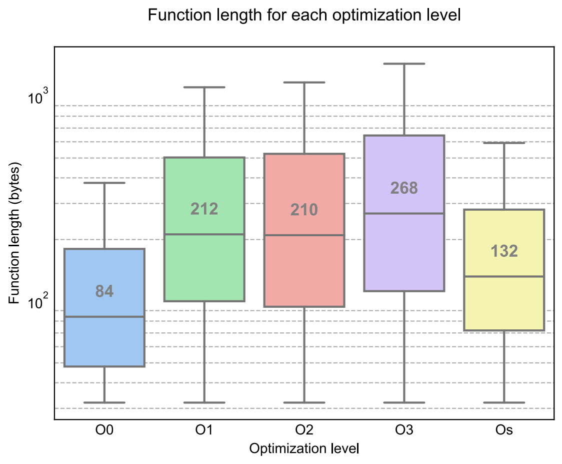


FIGURE 13. Statistics about the length of $47 \cdot 10^6$ functions. The value inside each box refers to the median.

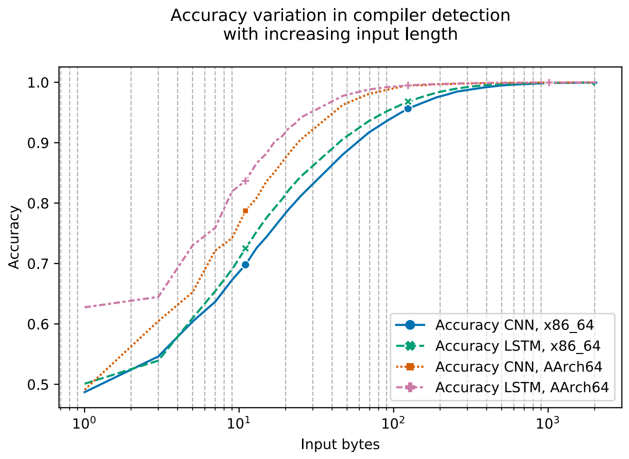


FIGURE 14. Accuracy in the compiler detection.

TABLE 5. Median number of bytes per function for each optimization level in each architecture. Results collected over a total of $47 \cdot 10^6$ functions.

Arch.	O0	O1	O2	O3	Os
x86_64	65	223	220	257	125
AArch64	68	232	220	260	132
RISC-V	84	160	166	218	113
SPARC	104	192	220	268	132
PowerPC	136	224	264	292	148
MIPS	184	276	284	376	188
ARM	72	188	174	232	106

TABLE 6. Accuracy of each optimization level limiting input to the median number of bytes per function at that optimization level. Results obtained with the LSTM.

Arch.	O0	O1	O2	O3	Os
x86_64	0.992	0.935	0.568	0.579	0.761
AArch64	0.984	0.948	0.719	0.590	0.694
RISC-V	0.987	0.863	0.729	0.344	0.791
SPARC	0.985	0.899	0.829	0.512	0.808
PowerPC	0.988	0.968	0.701	0.527	0.884
MIPS	0.985	0.986	0.786	0.771	0.909
ARM	0.984	0.951	0.658	0.529	0.807

medians for all the architectures together, more precise results for each architecture are reported in Table 5.

At this point, we calculated the accuracy for each of the listed medians. Figure 10 and 11 show the overall accuracy of correctly predicting each optimization level. Instead, we want to consider the accuracy of predicting each optimization level at its own statistical median, that represents the typical length of a function at that optimization level. As an example, for architecture x86_64 we evaluate the accuracy for O0 with an input of 65 bytes, for O1 with an input of 223 bytes. Table 6 shows the accuracy at these input lengths.

The Table confirms the results we obtained in Section V-A. Optimization levels O0 and O1 are easy to detect even at function granularity. The same goes for Os, although with a lower accuracy. The problem is, again, distinguishing between O2 and O3, as the median length of each function at that optimization level is not enough for an accurate prediction.

Regarding the compiler detection, the accuracy plot for increasing number of bytes can be seen in Figure 14. Unlike the CNN and LSTM comparison for optimization level detection, in Figure 12, we can see very few differences between the two networks, even with shorter inputs. Moreover, the accuracy is high even when there is not much data available; for example, with only 100 bytes, it is possible to have more than 90% accuracy. This means that we can correctly predict the compiler, even with function granularity.

Given these results, we can answer RQ_{min} as follows:

When performing a function grained analysis, with a short input, it is generally possible to detect O0, O1 and Os optimization level. O2 and O3, instead, requires as much bytes as possible, given their subtle differences. In contrast, compiler detection does not suffer this problem, achieving great accuracy even with 10^2 input bytes.

C. ENCODING

After showing how function-grained analysis is possible for some flags in Section V-B, we want to explore a possible improvement by removing redundant information from the input array. This stems from the conclusion of Chen *et al.* who found removing x86_64 prefixes increases the accuracy [11]. In this section, we compare the raw input with the encoded variant which is explained in Section IV-B. The result of this analysis is shown in Figure 15, depicting only the x86_64 architecture.

We can note how the encoded variant reflects the same difference between LSTM and CNN previously highlighted in Section V-B, in particular in Figure 12. More interestingly, the encoded variant reaches its maximum accuracy with an input length of approximately 250 bytes. The raw input variant, instead, as more bytes are supplied to it, steadily improves in accuracy beyond this limit.

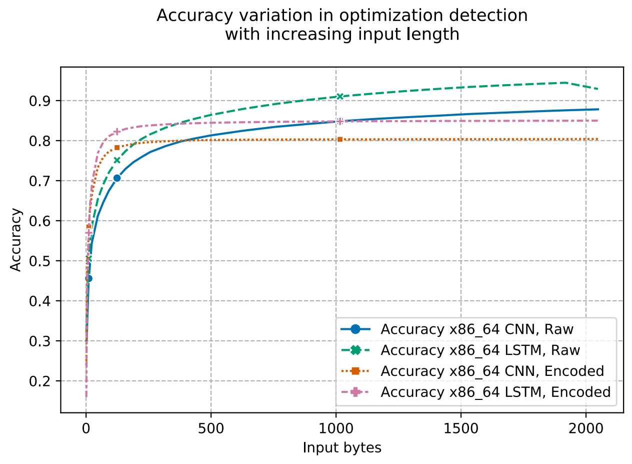


FIGURE 15. Accuracy in the optimization detection with encoded input data.

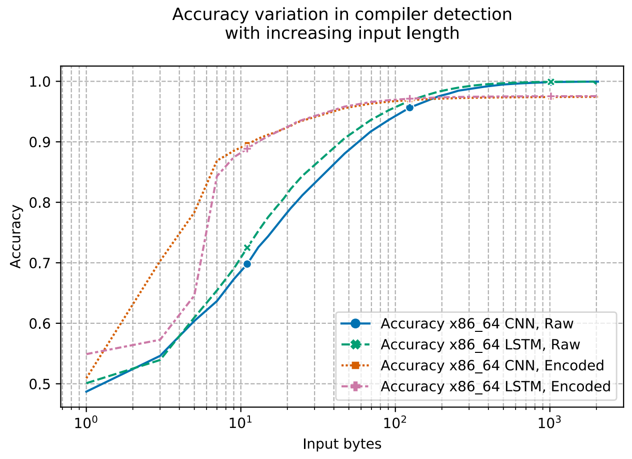


FIGURE 16. Accuracy in the compiler detection with encoded input data.

Moreover, the comparison we used in the Figure is biased towards the encoded variant, especially when we use a small number of bytes. In fact, for any given number of bytes in the encoded variant, we have more bytes available in the corresponding raw one. We calculated this difference to be an average of 186 bytes per function.

A similar situation can be seen when analyzing compiler detection, as depicted in Figure 16. In this case, we can see that the encoded variant reaches maximum accuracy at approximately 100 bytes without further improvements. In contrast, the raw data accuracy continues to increase, outperforming the encoded variant at 150 bytes and peaking at 1000 bytes.

We decided, however, against extending this analysis to all seven architectures. In fact, in Section I, one of the motivations of our study was to have an automated way of detecting optimization flags that does not require deep knowledge of the underlying architecture. To generate the encoded variant, however, it is necessary to possess a basic knowledge of the target architecture, which contradicts our original motivation. This fact, in addition to the poor performance and the need for

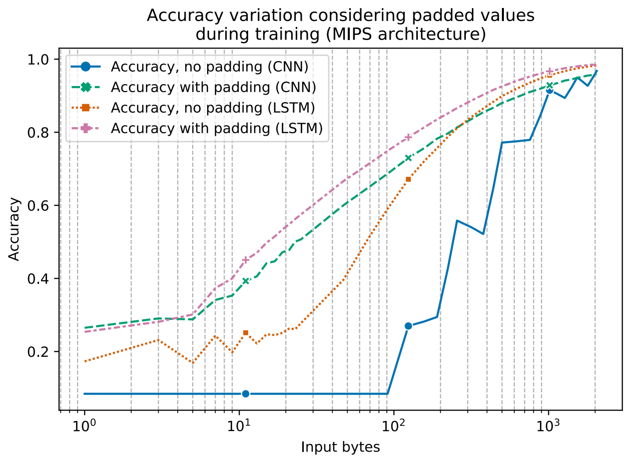


FIGURE 17. Accuracy variation in the optimization detection evaluation when including padding data during training.

accurate disassembly prompted us to abandon the encoded variant study.

Before concluding this section, it is worth noting that the study of Chen *et al.* used a dataset 100 times smaller and determined the encoding variant was remarkably better [11]. In our previous study, we used a dataset 10 times smaller than our current dataset and determined the encoding variant to be on par with the raw variant [12]. We can easily assume that with a smaller dataset, the network is less capable of learning which information is useful and what is not in the raw data. This would explain why in previous studies the encoded variant, which provides data without useless prefixes, was more competitive. However, with a sufficiently large dataset, the encoded variant does not offer any advantages.

We can thus conclude $RQ_{encoding}$ as follows:

Disassembling and encoding the data does not provide additional benefits, requiring knowledge of the underlying architecture and function disassembly for an overall lower accuracy.

D. PADDING

In Section IV-B we assert that our networks perform worse if, during training, raw byte sequences are never padded during training, and then padded sequences are predicted. In this section we present RQ_{pad}, and investigate the difference between padding during training and not padding. In this experiment, we trained two networks with the same dataset, seed, and samples ordered in the same way. However, in one case; the training values were always of 2048 bytes, in the other case, they were randomly cut in the interval [32, 2048], following the distribution explained in Section IV-C.

We evaluated both CNN and LSTM over the MIPS architecture, which achieved the best results. The differences between the padded and unpadded variants are shown in Figure 17

From the figure, we can see how the absence of padding during training is a problem when evaluating small input

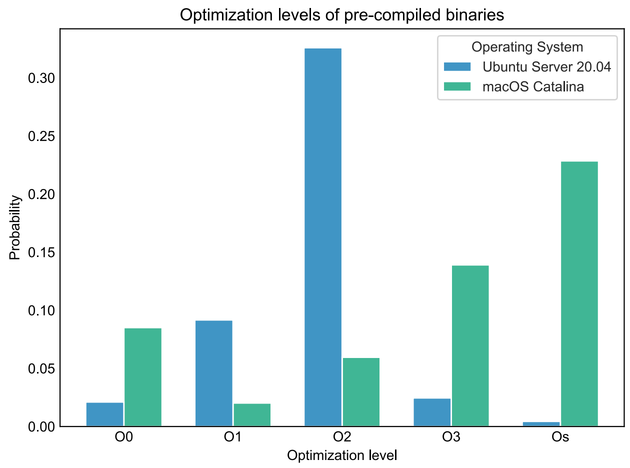


FIGURE 18. Optimization level in pre-compiled binaries shipped with Ubuntu server 20.04 and macOS Catalina.

vectors. For example, the LSTM is approximately 10% less accurate and reaches its counterpart trained with padding only when the input vectors are longer than 1000 bytes. The CNN results are even more extreme: with less than 100 bytes the network trained without padding always predicts the same output, and even at 125 bytes, there is a 60% accuracy gap between the two variants.

Although not presented, we performed this analysis also in the `x86_64` architecture, and obtained similar results.

We can thus conclude RQ_{pad} as follow:

If inputs are never padded during training, networks will have significantly lower accuracy while predicting shorter sequences.

E. OCCURRENCE

To conclude the evaluation, we would like to provide some data on the optimization flag distribution in a real scenario. To do so, we took every binary and library inside an *Ubuntu Linux 20.04 server* and *macOS 10.15.7 Catalina*, both unmodified. For each binary, we predicted the optimization level using our LSTM model for `x86_64`, reporting the results.

To analyze each binary, we divided the binary into several chunks of 2048 bytes, which is the same as our max input length. We then performed the inference for each chunk, and calculated an average between all the chunks. For each chunk, we weighted its contribution to the average by the network accuracy achieved at the chunk's input length.

Figure 18 shows the results of this analysis, performed over 10254 and 1216 files, respectively, for Ubuntu and macOS. Given the highly imbalanced number of input files, the histogram was normalized.

We can note how the distribution of files in the Linux system tends towards the O2 optimization level. This is not surprising, as the O2 optimization level provides the highest optimization without increasing the code size to the same extent as O3. The latter, in fact, could generate a larger code that does not fit in the instruction cache, resulting in overall slower execution [35]. Therefore, O2 is the suggested optimization level in some distributions, such as Gentoo Linux.[9] The macOS result, despite being more diverse, shows how most of its core programs are optimized for code size. As this is rather uncommon, we verified this results by manually inspecting the publicly available build scripts for Apple software.[10] As a confirmation, in most Makefiles we can find Os as the default optimization flag, explaining the histogram results. However, no reason for this choice is given in the build scripts.

Nonetheless, this final analysis is useful to prove the point that targeting binary analyses at the O2 optimization level assuming it to be the default one [7] may be a completely wrong assumption in some cases.

With this data, we can answer $RQ_{occurrence}$ as follows:

The most common optimization found in Ubuntu Linux is O2. In contrast, on macOs, Os is more common although not as dominant as O2 is for the Linux case. This proves that expecting O2 to always be the most common optimization level may be a false expectation.

VI. DISCUSSION

The results obtained in Section V provide a fast way of detecting the optimization level with multiple granularities.

One major change from previous studies is the lack of disassembly in our approach. Using a disassembler in order to retrieve function boundaries can be very time consuming. For large binaries, if the function grain is not required, previous approaches still require functions beginning and function ending, where ours can select a random 2048 bytes from the `.text` section and be dominated by the inference time. Unlike disassembly, dumping raw bytes from an executable or library is very fast, because it involves just reading the file itself.

If a function grain is necessary, the function headers may be retrieved by other means (i.e. Deep Learning). Given that our method does not require to preprocessing of the input data, we can skip disassembly also in this case, avoiding again the slowest part of binary analysis. This allows our tool to be used to check the compilation flags even at runtime, without any noticeable performance impact.

This lack of disassembly is the result of our evaluation in Section V-C, which contradicts the claim of Chen *et al.* that the encoded variant is better [11]. As mentioned in the section, this could be due to our larger dataset, which allowed the network to automatically learn which data is useful in the input architecture and which is not, rendering manual encoding useless.

In our study, given the larger size of our analysis, we also focused on producing an automated script to generate the

[9]https://wiki.gentoo.org/wiki/GCC_optimization
[10]http://opensource.apple.com

dataset. In fact, manually compiling a matrix of five optimization levels using seven architectures would have required an unmanageable amount of time. In addition, with this automated generation, we can extend the study to additional flags with small changes in the scripts parameters. In previous approaches, including ours, the entire dataset had to be manually regenerated to add new flags, the most tedious part of this entire study.

In addition, thanks to the small number of bytes required by our method, we can target very small portions of code, and thus, it can be used to check which portions of the binary match the used compilation flags. This allows for better categorization of the binary content and can help in binary analysis. In fact, if a small portion of the file is found with different flags or compiler than the rest of the file, there is a high probability that this portion belongs to a static library or a different compilation unit.

VII. LIMITATIONS AND FUTURE WORKS

The analysis we performed was limited to a pair of compilers and the most common optimization levels. In this study, we have shown that detection results can vary greatly between different architectures, and we have no guarantee that this analysis can be extended to more architectures without sacrificing accuracy. This is true even in the case of different compilers. The main difficulty in our study was distinguishing between O2 and O3 given their similarities. However, optimization levels are compiler specific and we cannot assume compilers other than GCC or Clang provide the same set of optimization levels.

Moreover, in this study, we focused entirely on optimization levels instead of specific flags. Although it would be easy to consider optimization flags, given our automated dataset generation, the classification should probably change from multiclass to multilabel. Furthermore, some flags would be challenging if not impossible to detect, the ‘‘dead code elimination’’ flag being one example.

Future work will involve assessing the feasibility of this multilabel classification, especially in compilers other than GCC or Clang.

VIII. CONCLUSION

In this paper, we have described two deep learning networks, one based on a long short-term memory model and the other based on a convolutional neural network model. We evaluated them in seven different architectures and showed that they can achieve between 92% and 98% accuracy while detecting between five different optimization levels and over 99.95% accuracy while detecting two different compilers.

We also provided an evaluation of the minimum number of bytes needed for accurate predictions, combined with statistical data about the different architectures and their median function length for each optimization level. Ultimately we proved that function grained optimization level detection is possible unless we are not aiming to distinguish between O2 and O3.

The results obtained are consistent with the initial motivation for our study: when comparing the structure of different binaries we reported the highest accuracy drop emerging in the case of different compilers or O0 compared with any other optimization level. These are also the values with the highest detection accuracy in our study, suggesting that our approach may be useful in detecting accuracy drop when comparing different binaries.

IX. REPLICATION

The dataset used in our study can be found on Zenodo at the following url [25]. This dataset contains each binary, divided by architecture, optimization level and compiler. Source code and pre-trained models can be found publicly on GitHub.[11]

REFERENCES

[1] K. Hoste and L. Ee Khout, ‘‘Cole: Compiler optimization level exploration,’’ in *Proc. 6th Annu. IEEE/ACM Int. Symp. Code Gener. Optim.*, Apr. 2008, pp. 165–174.
[2] M. Demertzi, M. Annavaram, and M. Hall, ‘‘Analyzing the effects of compiler optimizations on application reliability,’’ in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Nov. 2011, pp. 184–193.
[3] Y. David, N. Partush, and E. Yahav, ‘‘Similarity of binaries through re-optimization,’’ in *Proc. 38th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2017, pp. 79–94.
[4] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, ‘‘Ramblr: Making reassembly great again,’’ in *Proc. NDSS*, 2017, pp. 1–15.
[5] N. E. Rosenblum, B. P. Miller, and X. Zhu, ‘‘Extracting compiler provenance from program binaries,’’ in *Proc. 9th ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng.*, 2010, pp. 21–28.
[6] J. Pallister, S. J. Hollis, and J. Bennett, ‘‘Identifying compiler options to minimize energy consumption for embedded platforms,’’ *Comput. J.*, vol. 58, no. 1, pp. 95–109, Jan. 2015.
[7] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, ‘‘Cross-architecture bug search in binary executables,’’ in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 709–724.
[8] N. Rosenblum, B. P. Miller, and X. Zhu, ‘‘Recovering the toolchain provenance of binary code,’’ in *Proc. Int. Symp. Softw. Test. Anal.*, 2011, pp. 100–110.
[9] S. Hochreiter and J. J. Schmidhuber, ‘‘Long short-term memory,’’ *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.
[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, ‘‘Imagenet classification with deep convolutional neural networks,’’ in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.
[11] 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, Switzerland: Springer, 2019, pp. 35–47.
[12] D. Pizzolotto and K. Inoue, ‘‘Identifying compiler and optimization options from binary code using deep learning approaches,’’ in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2020, pp. 232–242.
[13] O. Katz, Y. Olshaker, Y. Goldberg, and E. Yahav, ‘‘Towards neural decompilation,’’ 2019, *arXiv:1905.08325*.
[14] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas, ‘‘Malware classification with recurrent networks,’’ in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Apr. 2015, pp. 1916–1920.
[15] B. Athiwaratkun and J. W. Stokes, ‘‘Malware classification with LSTM and GRU language models and a character-level CNN,’’ in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Mar. 2017, pp. 2482–2486.
[16] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, ‘‘Compiler optimization-space exploration,’’ in *Proc. Int. Symp. Code Gener. Optim.*, Mar. 2003, pp. 204–215.
[17] Z. Wang and M. O’Boyle, ‘‘Machine learning in compiler optimization,’’ *Proc. IEEE*, vol. 106, no. 11, pp. 1879–1901, Nov. 2018.

[11] http://github.com/inoueke-n/optimization-detector

[18] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Comput. Surv.*, vol. 51, no. 5, pp. 1–42, Jan. 2019.

[19] 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 *Proc. Int. Symp. Code Gener. Optim. (CGO)*, Mar. 2006, pp. 1–22.

[20] 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, May 2003.

[21] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to recognize functions in binary code," in *Proc. 23rd USENIX Secur. Symp.*, 2014, pp. 845–860.

[22] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 611–626.

[23] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 99–116.

[24] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, "Debin: Predicting debug information in stripped binaries," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 1667–1680.

[25] D. Pizzolotto and K. Inoue, "Binary software compiled for different architectures with different optimization levels," Tech. Rep., Apr. 2021, doi: 10.5281/zenodo.4659370.

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

[27] D. Andriesse, X. Chen, V. Van Der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *Proc. 25th USENIX Secur. Symp.*, 2016, pp. 583–600.

[28] M. Dwarampudi and N. V. S. Reddy, "Effects of padding on LSTMs and CNNs," 2019, *arXiv:1903.07288*.

[29] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*.

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

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

[32] V. Nair and G. E. Hinton, "Rectified linear units improve restricted Boltzmann machines," in *Proc. 27th Int. Conf. Mach. Learn. (ICML)*, 2010, pp. 807–814.

[33] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[34] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *J. Mach. Learn. Res.*, vol. 18, no. 1, pp. 6765–6816, 2017.

[35] M. T. Jones, "Optimization in GCC," *Linux J.*, vol. 2005, no. 131, p. 11, 2005.

DAVIDE PIZZOLOTTO received the B.S. degree in computer science and the M.S. degree in computer science from the University of Trento, in 2015 and 2019, respectively. He is currently pursuing the Ph.D. degree with the Graduate School of Information Science and Technology, Osaka University. Previously, he was a Research Assistant at Fondazione Bruno Kessler (FBK). His research interests include code obfuscation, binary code analysis, and source code analysis and transformation.

KATSURO INOUE (Member, IEEE) received the Ph.D. degree from Osaka University, in 1984. He was an Associate Professor with the University of Hawaii at Manoa, from 1984 to 1986. After becoming an Assistant Professor with Osaka University, in 1986, he has been a Professor, since 1995. His research interests include software engineering, especially software maintenance, software reuse, empirical approach, program analysis, code clone detection, and software license/copyright analysis.