Compiler and optimization level recognition using graph neural networks

Sébastien Bardin, Tristan Benoit, Jean-Yves Marion

Compiler and optimization level recognition using graph neural networks

Sébastien Bardin[1] , **Tristan Benoit**[2] and **Jean-Yves Marion**[2]
[1]CEA, LIST, Paris-Saclay, France,
[2]Université de Lorraine, CNRS, LORIA Nancy, France
sebastien.bardin@cea.fr, {tristan.benoit,jean-yves.marion}@loria.fr

Abstract

We consider the problem of recovering the compiling chain used to generate a given bare binary code. We present a first attempt to devise a Graph Neural Network framework to solve this problem, in order to take into account the shallow semantics provided by the binary code's *structured* control flow graph (CFG). We introduce a Graph Neural Network, called Site Neural Network (SNN), dedicated to this problem. Feature extraction is simplified by forgetting almost everything in a CFG except transfer control instructions. While at an early stage, our experiments show that our method already recovers the compiler and the optimization level provenance with very high accuracy. We believe these are promising results that may offer new, more robust leads for compiling tool chain identification.

1 Introduction

The problem. Identifying the *compiling chain*, i.e. both the compiler (e.g. Visual Studio) and its optimization options (e.g. $-O1$, $-O2$), that have been used to produce a given bare binary code is an important problem in at least two scenarios: to determine security flaws inside binary codes and to identify known functions.

- Applications are built by linking together libraries that are quite often commercial off-the-shelf (COTS)[1]. As a result, applications are developed more easily and quickly, and are usually more robust since COTS are normally already well-tested components. On the other hand, developers do not have the source code of COTS and in fact they do not even know the compiling chain used to generate these COTS. This is an important issue in software maintenance and long-term support. Indeed, compilers may inject vulnerabilities that are discovered after the COTS released and after the deployment of the applications that used them. For example, CVE-2018-12886 describes a vulnerability allowing an attacker to bypass stack protection and [Hohnka *et al.*, 2019] is a recent comprehensive study on vulnerabilities produced by compilers. Hence, there is a need to be able to retrieve the compiling chain in order to assess whether an application may present a certain vulnerability.
- The library function identification in a binary code is another primary issue for software maintenance and security, such as cloning detection (see for example [White *et al.*, 2016] using Deep Learning) or malware reverse engineering (see for example [Calvet *et al.*, 2012] using I/O relationship). The function name identification problem is readily solved when the binary code analyzed is a well-behaved binary code, that is when it contains enough information to disassemble it. For example, IDA disassembler proposes the F.L.I.R.T algorithm [Guilfanov, 2012] based on signature-patterns that recognizes functions and assigns a name to them, while ByteWeight [Bao *et al.*, 2014] constructs a weight prefix tree of the function prologue by machine learning to identify functions. These methods works well for regular binary codes, yet identification fails in many situations: when the library is unknown, when the code is slightly modified in such a way that its pattern does not match anymore, or when it is stripped or obfuscated. And in this context, the identification of the compiler together with the compiling options used to generate a binary code may help [Shirani *et al.*, 2017].

Actually, we tested IDA Freeware edition to determine the compiler used to generate a binary code. For this, we performed the test manually on 15 unstripped and stripped binary codes. On unstripped binary codes, IDA correctly distinguish MinGW binary codes from Visual Studio ones. In the case of stripped binary codes, IDA is not able to find the correct compiler and assigns each binary code to Visual Studio, probably as a default setting. Moreover and in all cases, IDA was unable to retrieve the compiling options.

Goal and approach. *The goal of the paper is to devise a machine learning based solution to the compiling chain identification problem.*

Rosenblum et al.'s pioneering work [Rosenblum *et al.*, 2011] introduced the problem and a first solution based on Support Vector Machine (SVM). Most of the recent works on this topic [Yang *et al.*, 2019; Rahimian *et al.*, 2015; Chen *et al.*, 2019; Massarelli *et al.*, 2019] rely on machine-

[1]More than 70% of commercial applications used COTS according to a Gartner study.

learning based approaches, taking their roots in Convolutional Neural Networks (CNN). Hence, the extracted features of a binary code are embedded as a text or as an image in order to be fed to a CNN. *But a program is not like a text or an image that can be projected into a regular Euclidean space.*

Claim. We claim that binary code semantics should be taken into account and the first model is given by its Control Flow Graph (CFG). That is why we suggest to use Graph Neural Networks (GNN) [Zhou *et al.*, 2018]. As a result, relationships between basic blocks (nodes of CFG) are taken into account in the graph embedding, and the weight of a CFG node depends transitively on its neighbors.

Contribution. Our paper makes the following contributions:

1. We develop a Graph Neural Network based framework that we call *Site Neural Network* (SNN) to determine the compiler family and the optimization level that generate a given binary code. The overall architecture is displayed in Figure 1 and Figure 2. The preprocessing is made as follows : a CFG is extracted from a binary code and then abstracted by just preserving the skeletal control flow. Then, the abstracted flow graph is chopped into fixed size that we call *sites*. This preprocessing step is fully automatic and unsupervised. Next, the Site Neural network takes the graph of all sites as input in order to classify the binary code. The overall architecture of our framework follows the approach of adapting Residual Neural Network (ResNet) to sites [Zhao *et al.*, 2018] and by refining the model with adaptive max pooling layers [Shervashidze *et al.*, 2011] to graphs.

   Our approach has at least two advantages.

   (a) Compared to prior works [Rosenblum *et al.*, 2011; Yang *et al.*, 2019; Rahimian *et al.*, 2015; Chen *et al.*, 2019; Massarelli *et al.*, 2019], the model is quite simple because it is reduced to sites. There are no instruction specific features and no focus on prologue/epilogue of binary functions. And so, we do expect that Site Neural Networks are more robust and generic.

   (b) From a methodological point of view, Site Neural Networks provides end-to-end graph based classifiers. As a result, decisions and classifications should be more easily based on the binary code semantics.

2. Most of the works in compiler provenance based on Machine Learning use datasets comprised of a set of binary *functions* generated by different compilers and optimization levels. From our point of view, this approach creates a bias as identifying function boundaries can be difficult (stripped binary code, obfuscated COTS) – this bias might be acceptable depending on the context.

   Thus, in our dataset we use full binary *codes* without any knowledge about functions and their localization. Our dataset consists of about 30,000 binary codes compiled from 30,000 different source codes with two compilers and three optimization levels. Thus, we can train and test our approach with six distinct and well-balanced sets of binary codes and all having a different code sources. It is for this reason that in the remainder of the paper, we will use the macro-average measures. The fact that our datasets are well-balanced is evidenced by a respective macro-average F1-Score of 0.973 and 0.996.

3. The identification covers two compilers, Visual Studio and MinGW, and three types of optimization O0, O1, and O2. We evaluated our system in terms of detection accuracy on a broad dataset composed of about 30000 binary codes. It makes accurate predictions with an overall F1-score of 0.978 with an estimated standard deviation of 0.0049. Also, we demonstrate the ability of our SNN-based approach to successfully recover the compiling chain on both un-stripped and stripped binary codes from our datasets.

Overall, we believe these are promising results that may offer new, more robust leads for compiling tool chain identification.

2 Related works

As we previously said, Rosenblum et al. in a series of two seminal papers [Rosenblum *et al.*, 2010; Rosenblum *et al.*, 2011] were the first to attempt to recover the compiler and compiler options using SVM – where features are composed of regular expressions (*idioms*) on the assembly program together with 3-vertex graphlets. Rahimian et al. [Rahimian *et al.*, 2015] developed BinComp that uses a complex model based on three layers and where the last one is an Annotated Control Flow Graphs. All these features are embedded into a vector by applying a neighbor hash graph kernel.

More recently, three papers were published on compiler provenance. Yang et al. [Yang *et al.*, 2019] extracts 1024 bits from the object file and process them with a one-dimensional CNN. Chen et al. [Chen *et al.*, 2019] describes the Himalia approach based on a two-tier classifier. Features are extracted from binary functions and consist of a sequence of instruction types of fixed size, which are eventually completed by padding. Thus, Himalia focuses on prologue and epilogue of functions and as a result is able to explain its classification. That said, the authors made the strong hypothesis to be able to determine the function prologue and epilogue.

Lastly, the closest related work is by Massarelli et al. [Massarelli *et al.*, 2019]. They propose a graph embedding neural network based on methods developed in the field of natural language processing (NLP). The preprocessing is composed of two stages. The first stage transforms a sequence of instruction in a vector by measuring different instruction parameters. Then, the overall CFG is embedded into a graph, which is aggregated by a 2-round processes using a Recurrent Neural Network.

3 Background

Recent work used convolutional neural networks to predict the compiler toolchain. Two difficulties here are that (i) binary code semantics, denoted by a control flow graph, has to be taken into account and (ii) the input to the classifier, a binary code or CFG, can be of arbitrary size. The second

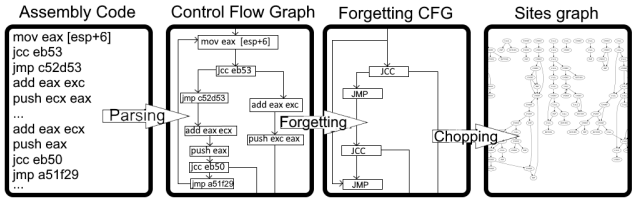

Figure 1: Overall architecture of the preprocessing phase

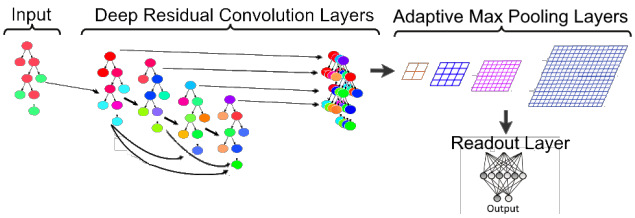

Figure 2: Overall architecture of Site Neural Networks

difficulty is yet inherent to most machine learning pipelines that can only handle inputs of a fixed size. In this paper, we replace convolutional neural networks (CNN) by graph neural networks (GNN) [Micheli, 2009; Zhou *et al.*, 2018]. The architecture of GNN is mostly based on a transformation of classical CNN architecture and for example the paper [Zhao *et al.*, 2018] generalizes ResNet [He *et al.*, 2015] to graphs. Inputs of a graph neural network is the representation of a graph and the survey [Hamilton *et al.*, 2017] discusses the different graph representation. Compared to unstructured data like texts (one-dimensional data) or images (two-dimensional data), the graph encoding must preserve certain properties like the shape or the connectivity.

A key feature is the pooling method. A pooling layer of GNN does not depend on the input size. For this, the pooling can just take the sum of node values. There are other methods such as sort pooling selecting a fixed sized set of maximum node values [Zhang *et al.*, 2018] or such as adaptive max pooling by dividing the matrix at each convolution in a fixed number of parts [Yan *et al.*, 2019], as illustrated in Figure 4.

Note that there is still a fundamental limitation in the GNN approaches. Indeed, subgraph isomorphism is an important problem in graph classification. In [Xu *et al.*, 2018], it is demonstrated that GNN cannot do better than WL-test of isomorphism [Weisfeiler and Leman, 1968], even if naming each node with an identifier increases the predictive capacity of the model as shown in [Seidel *et al.*, 2015].

4 Our method for compiler identification

4.1 Binary code preprocessing

The architecture of the preprocessing is shown in Figure 1. Inputs are binary codes. Each binary code is disassembled and the Control Flow Graph is built. Then, there are two more steps that we call the *forgetful phase* and the *chopping phase* that are explained below:

Symbol	Signification
RET	return
CALL	function call
JMP	unconditional jump
HLT	interruption
INVALID	failure when disassembling
UNDEF	unknown address
JCC	conditional jump
SWITCH	jump to multiples destinations

Table 1: Automatic labelling

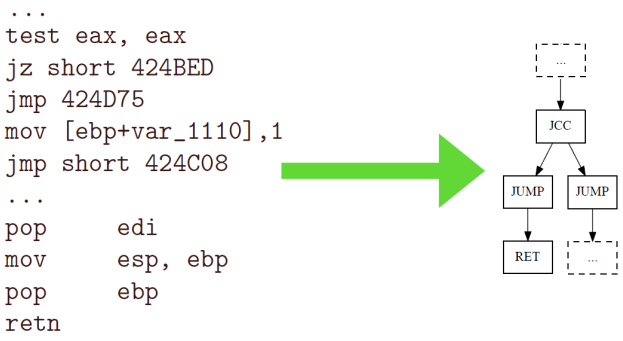

Figure 3: The forgetful phase

4.2 The forgetful phase

The forgetful phase consists in simplifying a CFG by removing sequential instructions and by just keeping control flow instruction types. Figure 3 illustrates this reduction. For this, the phase runs in two stages :

1. All consecutive nodes labeled by a sequential instruction (like `mov` or `add`) are pruned in one single node that is removed.
2. All remaining nodes are relabeled based on the instruction type following Table 1.

The forgetful phase respects the underlying structure of the input CFG and maps it to a reduced graph that we call the *forgetting CFG* for convenient notation in the remainder.

4.3 The chopping phase

The chopping phase cuts a forgetting graph into a set of small disconnected subgraphs. These subgraphs are called *sites* and their size is at most of 24 nodes. Sites are obtained using a breadth first search algorithm from the forgetting graph – the algorithm is presented in Appendix (Algorithm 1). We associate to each node of a site two features composed of the instruction type (see Table 1) and a unique identifier in order to solve the problem of anonymity.

Clearly, the forgetful and chopping phases reduce drastically the input dimension. Moreover, since each site has a small diameter at most 24), a small number of convolutions allows information to pass through all nodes. Notice that

sites are directed graphs, but they are processed as undirected graphs during convolution computations.

4.4 Site Neural Networks

The input are graphs that are built by the two previous phases from a binary code. Take a graph composed of a set of nodes V and represented by the adjacency matrix A. We define X_0 as the matrix containing the nodes attributes, thus it has a dimension of $n \times 2$.

Mini-batches. A single input is a set of sites which are collectively regrouped in a graph. In the training phase, the input graphs are partitioned into mini-batches. This gives us the opportunity to normalize the data [Ioffe and Szegedy, 2015]. Each mini batch B is normalized by calculating $\text{batchNorm}_B(x) = \frac{x-\mu_B}{\sigma_B}$, where μ_B is the observed mean and σ_B is the observed variance. Notice that, the observed means and variances are memorized, because they are reused in test time. This process has been shown to be successful, but is not yet understood in theory. The activation function is the rectified linear unit $\text{relu}(x) = \max(0, x)$.

Deep Convolution. Now, the vector sequence of node values $(Y_{k+1})_{k\geq 0}$ obtained after $k+1$ convolution(s) is defined as follows:

$$Y_1 = \text{relu}(\text{batchNorm}_B((A+I)X_0W_0 + b_0))$$
$$Y_{k+1} = (\text{relu}(\text{batchNorm}_B((A+I)Y_kW_k + b_k))|Y_k)$$

The notation $|$ is the matrix augmentation. As usual with deep convolutions, the output of one step is fed into every future step.

Dimensions. Let d_t be the hyper parameter corresponding to the second dimension of the matrix W_t at convolution t. The first dimension of the matrix W_k, for $k > 0$, is $\sum_{t=0}^{k-1} d_t$.

Output. The dimension of the matrix Y_k, for $k > 0$, is $n \times \sum_{t=0}^{k} d_t$ where n is the number of nodes. We now perform a pooling, which reduces the matrix of the last convolution to some smaller fixed size matrix.

Adaptive Max Pooling Layer(s). Following [Shervashidze *et al.*, 2011], a crucial point in our approach is the extraction of features based on the Weisfeiler-Lehman test of graph isomorphism. For this, we apply on the result of the convolutional layers an Adaptive Max Pooling (AMP) step. This pooling operation is defined by an operator $\text{amp}_{n,m}$ that reduces a matrix to a matrix of smaller dimension $n \times m$ as follows: Take a matrix M of dimension $u \times v$. M is cut into $n \times m$ matrices of kernel size $\lceil \frac{u}{x} \rceil \times \left\lceil \frac{v}{y} \right\rceil$. We take the maximum in each block. The figure 4 illustrates the adaptive max pooling computation. We iterate four times the adaptive max pooling operation to extract a fixed size representation of X.

Readout layer. At this point we have obtained from the adaptive max pooling layers a fixed size representation of our graph. The output of the adaptive max pooling layers is fed into a multilayer perceptron to predict the probability distribution of the class that the input graph should belong to.

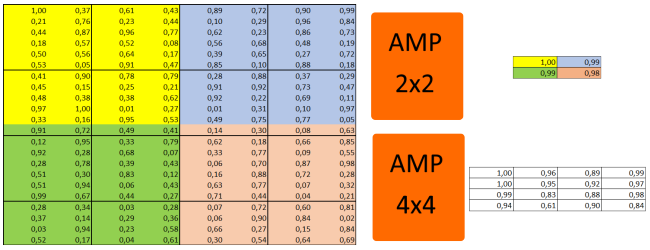

Figure 4: An example of two adaptive max pooling on a matrix of dimension 8×22. The first adaptive max pooling is of dimension 2×2, it has a kernel size of dimension 4×11. The second adaptive max pooling is of dimension 4×4, it has a kernel size of dimension 4×6.

5 Evaluation

5.1 Implementation details

We perform four convolutions with a dimension of 8 at each step. The hyper-parameter d_t of the matrix W_t is 8 for each convolution t. We use four pooling layers with $\text{amp}_{2,2}$, $\text{amp}_{4,4}$, $\text{amp}_{8,8}$ and $\text{amp}_{16,16}$ operators. The multilayer perceptron has four layers of respective numbers of weights of 384, 256, 128 and 2. We implemented our framework using the language python along with the machine learning library PyTorch. Prepossessing is performed using C++.

5.2 Datasets

We evaluated the performance of our framework against two datasets coming from two separate domains :

The Codeforces Dataset. This dataset is made from 19,986 source codes, that solve 91 problems from Codeforces [2]. We compiled them using Visual Studio 2019 and MinGW on Windows 10 with optimization options O0, O1, O2 and O3. As a result, we get 7 distinct classes of approximately 3,000 binary codes;

The CSmith Dataset This dataset is produced using CSmith [Yang *et al.*, 2011], a random program generator created to find bugs in compilers. We compiled 10,562 binaries, half of them with Visual Studio 2019 and the other half with MinGW using an optimization level among O0, O1, O2 and O3. The average size of each binary code is approximately 30Kb and there is at most 10 nested loops;

Similarities. In Codeforces Dataset, there are huge similarities between programs compiled with MinGW O2 and MinGW O3. Over a random sample of 2920 binary codes, 561 binary codes compiled with O2 are identical when compiled with O3. This has already been reported in [Egele *et al.*, 2014] and has been an issue to Chen et al. [Chen *et al.*, 2019] in their compiler optimization detection framework. Straightforwardly, this is not the case however in the Visual Studio Compiler which has only one advanced optimization level.

5.3 Research questions

We investigate the following research questions in order to validate our framework and to see its current limits:

[2] https://codeforces.com/

RQ1 Does our framework have the capacity to predict the compiler and optimization level of binary codes coming from our datasets?

RQ2 Using our framework, can a model learned from a specific dataset be applied on another dataset? The question is to know whether or not what has been learned with one dataset by our framework can be exploited on another distinct dataset.

RQ3 Are the performances decreasing when binary codes are stripped ? That is, take a SNN trained with an unstripped dataset: are we still able to detect the compiler and the compiling options from a stripped binary code?

5.4 RQ1: Compiler and optimization option identification

Methodology. To answer the first question, we consider both datasets separately. Due to the similarity between programs produced by MinGW -O2 and by MinGW -O3, we will consider MinGW -O2/O3 as a proper sub class. Figure 5 presents our hierarchical classifier. A site neural network is specialized to make the separation between two choices (e.g. between MinGW -O0/O1 and MinGW -O2/O3).

To run this experiment, each dataset is split in a train set, a validation set and a test set with resp. 80%, 10%, and 10% of the starting datasets. Next, for both datasets, each specialized site neural network is trained using 0.0005 as the learning rate during 30 epochs. The loss function is . And, we select the model for each specialized site neural network with the best accuracy on the validation split on some epoch.

After training, each site neural network is evaluated thanks to the validation set of the corresponding dataset. We performed this operation twice to mitigate randomness. Considering any class as important as any other, we choose to report the macro average of the precision, recall and F1-Score using three significant digits.

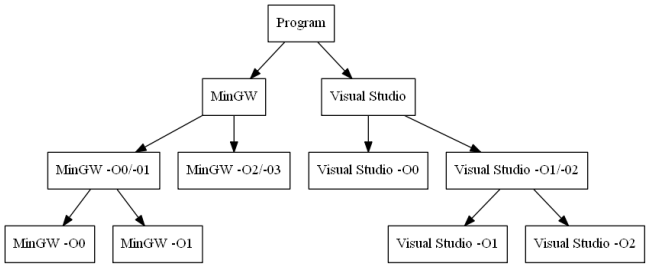

Figure 5: The first hierarchical classifier using five different site neural network specialist.

Results. Table 2 reports the performance of a classifier trained on Codeforces Dataset. It achieves an overall F1-score of 0.973 with an estimated standard deviation of 0.0017.

In table 3, we just report, due to the lack of space, the performance of a classifier trained on CSmith Dataset. It achieves a slightly better overall F1-score of 0.996 with an estimated standard deviation of 0.0023.

Class	Precision	Recall	F1-Score	Support
MinGW O0	0.960	0.961	0.961	285
MinGW O1	0.954	0.947	0.951	285
MinGW O2/3	0.994	0.997	0.996	570
VS O0	0.981	0.974	0.977	285
VS O1	0.975	0.975	0.975	285
VS O2	0.974	0.979	0.976	285
Macro Avg	0.973	0.972	0.973	1995

Table 2: Prediction on Codeforces Dataset.

	Precision	Recall	F1-Score	Support
Macro Avg	0.996	0.996	0.996	1050

Table 3: Prediction on CSmith Dataset

	Precision	Recall	F1-Score	Support
Macro Avg	0.979	0.978	0.978	3052

Table 4: Prediction on the full Dataset

In table 4, we report the performance of a classifier trained with the full Dataset. The full dataset is composed of the union of both datasets. It achieves an overall F1-Score of 0.978 with a standard deviation of 0.0049. Taking into account that the full Dataset contains one third instances from the CSmith Dataset and two third from Codeforces Dataset, we could have expected an overall F1-Score of $0.996 \times \frac{10562}{30548} + 0.973 \times \frac{19986}{30548} \approx 0.981$. Thus, there is a small loss of 0.003.

Conclusion. We conclude that our framework has an elevated capacity to predict a compiler and the optimization level of a binary from our datasets. Our framework exhibits a potential loss when the diversity of the input data increases. Further analysis is needed to tackle a more robust statistical analysis.

5.5 RQ2: Applying a model learned on a dataset to a new dataset

We used our previous models on the datasets we had not learned.

Methodology. We conduct two symmetric experiments. First, we trained specialized site neural networks as explained previously following the decision tree in 5 with Codeforces dataset. Then, we evaluate trained specialized site neural networks with Codeforces dataset on CSmith dataset in order to see if the starting dataset is sufficient.

Second, we inverse the role of Codeforces dataset and of CSmith dataset. That is, we evaluate trained specialized site neural networks with CSmithdataset on Codeforces dataset.

	Precision	Recall	F1-Score	Support
Macro Avg	0.531	0.505	0.477	1050

Table 5: Prediction on CSMith using Codeforces models

	Precision	Recall	F1-Score	Support
Macro Avg	0.225	0.254	0.201	1995

Table 6: Prediction on Codeforces using CSmith models

	Precision	Recall	F1-Score	Support
Macro Avg	0.993	0.993	0.993	1995

Table 7: Prediction on Codeforces stripped binary codes

	Precision	Recall	F1-Score	Support
Macro Avg	0.999	0.999	0.999	1050

Table 8: Prediction on CSmith stripped binary codes

Results. In table 5, we report the performance of a classifier trained with Codeforces dataset on the CSmith Dataset. It achieves an overall F1-Score of 0.477 with an estimated standard deviation of 0.0191. In table 6, we report the performance of a classifier trained with the CSmith dataset on Codeforces Dataset which achieves an overall F1-Score of 0.201 with a standard deviation of 0.0537. We are intrigued by the fact that no instance was classified as a MinGW -OO compiler.

Conclusion. We speculate that due to the purpose of the CSmith generated programs, the CSmith Dataset is more specific than the Codeforces Dataset. While the result of predicting the CSmith dataset shows a clear loss of 0.519, we still achieve a moderate F1-score. We conclude that what have been learned with one dataset by our framework may partly be exploited on a distinct dataset.

5.6 RQ3 : Are the performances decreasing when binary codes are stripped ?

Methodology. We used both previous models that are trained on *un-stripped* binary codes. And we challenge them with stripped binary code of the same dataset. We selected 10% of binary codes to make both experiments.

Results. Results are given in Tables 7 and 8 respectively. In both cases, the F1-Score obtained in stripped binary is very similar to the one on un-stripped binary codes, that was presented in section 5.4, see Table 2 and Table 3.

Conclusion. When binary codes are stripped and unlike IDA, there is no noticeable difference in the accuracy of the recognition of the compilers and of the compiling option. This result confirms the fact that this approach can be used to detect the COTS compilation chain as part of software maintenance. This result confirms the fact that this approach may be used in detecting COTS compiling chain for software maintenance.

6 Limitations

- Our dataset is composed for one part of small programs (most file sizes are around 30kb) coming from Codeforces dataset and for the other part, synthetic programs generated by CSmith. It would be interesting to use a more diverse dataset. Nevertheless, our evaluation on both datasets at least demonstrates the accuracy of SNN.
- The experiments should be pursued to incorporate other compilers with more optimizations. We leave this for future work.
- Our approach was tested and validated on un-stripped and stripped binary codes. In adversarial contexts where binary are obfuscated or when we are dealing with malware, the situation is quite different. We believe this is a challenge worth working on.

7 Conclusion

Our starting hypothesis is that binary code is not unstructured data and that semantics is important. In this ongoing work, we begin to explore the possibility of (i) extracting semantic features in the form of graphs and (ii) processing the neural networks of the graphs in order to propagate the information according to the topology of the graph. The outcome is to improve binary code analysis in particular in the context of obfuscated codes and malware.

This work opens several immediate questions. The combination of the forgetful phase followed by the chopped phase provides a simple and realistic feature graph model. That said, one could think of a first phase that would leave out less information and various ways of cutting out a graph by having, for example, sites of different sizes. Also, the pooling layers play an important role. It should be worth looking at which features are useful and how they are intertwined in order to improve pooling. This question bounces off the question of semantics. Finally, CFG provides only a very shallow semantics view of a given program. An interesting question would be to automatically extract and take advantage of some sort of richer semantic features.

As a conclusion, we believe our preliminary results are promising and may offer new, more robust leads for compiling tool chain identification.

Acknowledgments

This work is supported by a public grant overseen by the French National Research Agency (ANR) as part of the "Investissements d'Avenir" French PIA project "Lorraine Université d'Excellence", reference ANR-15-IDEX-04-LUE.

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[Bao *et al.*, 2014] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium*, p. 845–860, 2014.

[Calvet *et al.*, 2012] Joan Calvet, José M. Fernandez, and Jean-Yves Marion. Aligot: Cryptographic function identification in obfuscated binary programs. In *Proceedings of the 2012 ACM CCS*, p. 169–182, 2012.

[Chen *et al.*, 2019] Yu Chen, Zhiqiang Shi, Hong Li, Weiwei Zhao, Yiliang Liu, and Yuansong Qiao. Himalia: Recovering compiler optimization levels from binaries by deep learning. In *Intelligent Systems and Applications*, p. 35–47. Springer Publishing, 2019.

[Egele *et al.*, 2014] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium*, p. 303–317, 2014.

[Guilfanov, 2012] Ilfak Guilfanov. Ida fast library identification and recognition technology (flirt technology): In-depth, 2012.

[Hamilton *et al.*, 2017] William L. Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. 2017. arxiv:1709.05584.

[He *et al.*, 2015] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[Hohnka *et al.*, 2019] Michael J. Hohnka, Jodi A. Miller, Kenrick M. Dacumos, Timothy J. Fritton, Julia D. Erdley, and Lyle N. Long. Evaluation of compiler-induced vulnerabilities. *Journal of Aerospace Information Systems*, 16(10):409–426, 2019.

[Ioffe and Szegedy, 2015] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[Massarelli *et al.*, 2019] Luca Massarelli, Giuseppe Luna, Fabio Petroni, and Leonardo Querzoni. Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. 2019.

[Micheli, 2009] Alessio Micheli. Neural network for graphs: A contextual constructive approach. *Neural Networks, IEEE Transactions on*, 20:498 – 511, 2009.

[Rahimian *et al.*, 2015] Ashkan Rahimian, Paria Shirani, Saed Alrbaee, Lingyu Wang, and Mourad Debbabi. Bincomp: A stratified approach to compiler provenance attribution. *Digital Investigation*, 14:S146 – S155, 2015.

[Rosenblum *et al.*, 2010] Nathan Rosenblum, Barton Miller, and Xiaojin Zhu. Extracting compiler provenance from program binaries. p. 21–28, 2010.

[Rosenblum *et al.*, 2011] Nathan Rosenblum, Barton P. Miller, and Xiaojin Zhu. Recovering the toolchain provenance of binary code. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, p. 100–110, 2011.

[Seidel *et al.*, 2015] J. Seidel, R. Wattenhofer, and Y. Emek. *Anonymous Distributed Computing: Computability, Randomization, and Checkability*. ETH-Zürich, 2015.

[Shervashidze *et al.*, 2011] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(77):2539–2561, 2011.

[Shirani *et al.*, 2017] Paria Shirani, Lingyu Wang, and Mourad Debbabi. Binshape: Scalable and robust binary library function identification using function shape. In *DIMVA*, p. 301–324. Springer Publishing, 2017.

[Weisfeiler and Leman, 1968] B. Yu. Weisfeiler and A. A. Leman. Reduction of a graph to a canonical form and an algebra arising during this reduction. 1968.

[White *et al.*, 2016] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM ASE*, p. 87–98, 2016.

[Xu *et al.*, 2018] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks?, 2018.

[Yan *et al.*, 2019] Jiaqi Yan, Guanhua Yan, and Dong Jin. Classifying malware represented as control flow graphs using deep graph convolutional neural network. In *49th Annual IEEE/IFIP DSN*, p. 52–63, 2019.

[Yang *et al.*, 2011] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN PLDI*, p. 283–294, 2011.

[Yang *et al.*, 2019] S. Yang, Z. Shi, G. Zhang, M. Li, Y. Ma, and L. Sun. Understand code style: Efficient cnn-based compiler optimization recognition system. In *ICC 2019 - IEEE*, p. 1–6, 2019.

[Zhang *et al.*, 2018] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *32th AAAI Conference on Artificial Intelligence*, 2018.

[Zhao *et al.*, 2018] Wenting Zhao, Chunyan Xu, Zhen Cui, Tong Zhang, Jiatao Jiang, Zhenyu Zhang, and Jian Yang. When work matters: Transforming classical network structures to graph cnn. *arXiv:1807.02653*, 2018.

[Zhou *et al.*, 2018] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications, 2018.

A Additional material

The breadth first search algorithm of the chopping phase performs a limited exploration of the graph. During that exploration, it disconnects from the graph multiple small subgraphs .

Algorithm 1 Graph chopping algorihtm

Input: A forgetting graph $G = (V, E)$, a root vertex r in V
Parameters: n the number of sites to extract, s max node in a site
Output: A graph containing a maximum of n sites

Let $G_r = (V_r, E_r)$ be a a graph.
while $|V| > 0$ and $n > 0$ **do**
  Let $q1$ be a queue.
  Let $q2$ be a queue.
  Let $g = (N, A)$ be a graph.
  push $q1, r$
  while $|N| < s$ **and** $|q1| > 0$ **do**
    empty $q2$
    for all $x \in q1$ **do**
      if $|N| > s$ **then**
        break
      end if
      for all y **such that** $(x, y) \in E$ **do**
        if $|N| > s$ **then**
          break
        end if
        if $y \in$ N **then**
          $A \leftarrow A \cup \{(x, y)\}$
          break
        end if
        $N \leftarrow N \cup \{y\}$
        $A \leftarrow A \cup \{(x, y)\}$
        push y in $q2$
      end for
    end for
    $q1 \leftarrow q2$
  end while
  $V \leftarrow V \setminus \{N\}$
  $E \leftarrow E \setminus \{(u, v) | u, v \in N\}$
  $r \leftarrow$ first vertex left in V
  $V_r \leftarrow V_r \cup N$
  $E_r \leftarrow E_r \cup A$
  $n \leftarrow n - 1$
end while
return G_r