Binary level toolchain provenance identification with graph neural networks

Tristan Benoit, Jean-Yves Marion, Sébastien Bardin

To cite this version:

Tristan Benoit, Jean-Yves Marion, Sébastien Bardin. Binary level toolchain provenance identification with graph neural networks. SANER 2021 - 28th IEEE International Conference on Software Analysis, Evolution and Reengineering, Mar 2021, Honolulu / Virtual, United States. pp.131-141, ⟨10.1109/SANER50967.2021.00021⟩. ⟨hal-03447628⟩

HAL Id: hal-03447628

https://hal.science/hal-03447628v1

Submitted on 24 Nov 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

Binary level toolchain provenance identification with graph neural networks

Tristan Benoit
Université de Lorraine, CNRS, LORIA,
F-54000 Nancy, France
tristan.benoit@loria.fr
,

Jean-Yves Marion
Université de Lorraine, CNRS, LORIA,
F-54000 Nancy, France
jean-yves.marion@loria.fr
,

Sébastien Bardin
CEA LIST
Université Paris-Saclay
Saclay, France
sebastien.bardin@cea.fr

***Abstract*—We consider the problem of recovering the compiling chain used to generate a given stripped binary code. We present a Graph Neural Network framework at the binary level to solve this problem, with the idea 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. To attain scalability at the binary level, feature extraction is simplified by forgetting almost everything in a CFG except transfer control instructions and performing a parametric graph reduction. Our experiments show that our method recovers the compiler family with a very high F1-Score of 0.9950 while the optimization level is recovered with a moderately high F1-Score of 0.7517. On the compiler version prediction task, the F1-Score is about 0.8167 excluding the clang family. A comparison with a previous work demonstrates the accuracy and performance of this framework.**

***Index Terms*—toolchain provenance, graph neural networks, binary code analysis**

I. Introduction

The problem Identifying the *toolchain provenance*, i.e. the compiler family (e.g. Visual Studio or GCC), the compiler version (e.g. 10.0, 12.0) and its optimization options (e.g. $-O1$, $-O2$), that have been used to produce a given stripped binary code is an important problem in at least two scenarios:

- *Determination of security flaws inside binary codes.* Applications are often built by linking together commercial off-the-shelf libraries (COTS)[1]. While allowing faster development cycles, developers do not have the source code of these COTS and do not know the compiling chain used to generate them. This is an important issue in software maintenance and long-term support as compilers may inject vulnerabilities that are discovered after the COTS released and after the deployment of the applications that used them [1]. For example, CVE-2018-12886 describes a vulnerability allowing an attacker to bypass stack protection in GCC 4.1 though 8. Hence, there is a need to be able to retrieve the compiling chain to assess whether an application may present a certain vulnerability;
- *Identification of known functions.* Library function identification in a binary code is another primary issue for software maintenance and security, such as clone detection (see for example [2] using Deep Learning) or malware reverse engineering (see for example [3] using I/O relationship). The function name identification problem is readily solved when the binary code under analysis is well-behaved, that is when it contains enough information to disassemble it. For example, IDA disassembler proposes the F.L.I.R.T algorithm [4] based on signature-patterns that recognizes known functions, while ByteWeight [5] constructs a weight prefix tree of the function prologue by machine learning to identify known functions. These methods work well for regular binary codes, yet identification fails in many situations: unknown libraries, slightly modified binaries, stripped or obfuscated binaries. Compiler chain provenance identification may help in such cases [6].

We tested IDA Freeware edition to determine the compiler used to generate a binary code. For this, we performed a test manually on 15 unstripped and stripped binary codes. A *stripped binary* is a binary code without debugging symbols. On unstripped binary codes, IDA correctly distinguishes 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 toolchain provenance identification problem over stripped binary codes.*

Rosenblum et al.'s pioneering work [7] introduced the problem and a first solution based on Support Vector Machine (SVM) over binary functions disassembled from a binary. Recent works on this topic [8], [9] rely on neural networks, either Convolutional (CNN) or Recurrent (RNN). The extracted features of a binary code are *embedded* as a text or as an image to be fed into a neural network. Rahimian et al. [10] takes a stratified approach: first, the compiler family is guessed

This work is supported by (i) 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, and (ii) has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 830927 (Concordia).

[1]More than 70% of commercial applications used COTS (Gartner).

and then, given the compiler family, the optimization level is determined. With a combination of features coming from the control graph and the instruction sequences, the toolchain provenance is attributed with hash kernels and fingerprints.

A recent work by Massarelli et al. [11] extracts binary function Control Flow Graph (CFG). Their neural network can be decomposed into two parts: first, the raw binary sequences in each basic block are dealt with natural language processing (NLP) techniques, second, the graph is operated using convolutions – aggregating blocks to classify each binary function.

Instruction embedding is a topic in itself. Different embeddings have been adapted from the natural language processing setting such as asm2vec [12], cross-architecture instruction embedding [13], or i2v [11].

Claim We claim that the decision process can be done at the binary *program* level rather than at the binary *function* level like in most prior approaches. Going away from precise block level binary semantics, we suggest using a *forgetting* control flow graph (CFG) of the whole binary program. That is why we propose using Graph Neural Networks (GNN) [14]. 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. In doing so, we take the *proviso* that a program is not like a text or an image that can be projected into a regular Euclidean space, and consequently, it is worth keeping the program structure, at least partially.

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, compiler version and the optimization level that generate a given binary code. The overall architecture is displayed in Figure 1 and Figure 2. We extract a CFG from a binary code and then abstract it by preserving only the skeletal control flow. Then, this *forgetting* flow graph is *chopped* into fixed-size sub-graphs (*sites*) whose size is a parameter noted α in the remainder of the text. This preprocessing step is fully automatic and unsupervised. Next, the Site Neural Network takes the graph of all sites as input to classify the binary code. The overall architecture of our framework follows the approach of adapting Residual Neural Network (ResNet) to sites [15] and by refining the model with adaptive max pooling layers [16] to graphs. We also propose to use hierarchies of multiple SNN experts making binary decisions (e.g. 'Clang 8.0' versus 'Clang 8.5').
   Our approach has at least three advantages.
   - Compared to prior works [7]–[11], the model is quite simple because it is reduced to sites. There is no instruction specific feature and no focus on function prologues/epilogues. So, we do expect that Site Neural Networks are more robust and generic.
   - 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.
   - Our framework can easily be adapted to different contexts: chopping is parameterized by the sub-graph size α, while our SNN hierarchy allows adding new experts in a modular way – for example to deal with a new compiler version or a new option.
2) Most prior works in toolchain provenance based on Machine Learning use as datasets a set of binary *functions* generated by different compilers and optimization levels. From our point of view, this approach creates a bias, which might be acceptable depending on the context. Indeed, it might be difficult to correctly identify function boundaries when we deal with obfuscated COTS or stripped binaries and the preprocessing time to find functions, for example in COTS, may be important. Moreover, most of the time, libraries, like DLL, are dynamically linked and so it is not necessary to determine the compiler toolchain for each function individually. *That is why we focus on binary codes*. Hence, our dataset consists of full stripped binary *codes* without any knowledge about functions and their localization.
3) Some study suffers from a limited dataset variety (e.g., only 18 compiler configurations in Rosenblum et al. [17]). Our study covers a large number of possible compiling toolchains on Linux and Windows (both 64-bit systems). Indeed, the identification covers 23 different compiler versions of four major compilers: Clang, GCC, MinGW, Visual Studio. We take into consideration four classes of optimization -$O0$, -$O1$, -$O2$/-$O3$, and -Os – for a total of 92 compiler configurations. We evaluated our system in terms of detection accuracy on a broad dataset composed of about $36,272$ stripped binary codes compiled from $36,272$ different source code with four compiler families, 23 different compiler versions, and five optimization levels. Thus, we can train and test our approach with 121 distinct and well-balanced sets of binary codes, all having different code sources.

We demonstrate that toolchain provenance identification *at program level* is feasible with good precision and accuracy together with an efficient learning phase – obtaining better classification results than the most recent prior function-level methods [11] together with much smaller learning time (Section VI-A). Overall, we believe these results are promising and may offer new, more robust leads for toolchain provenance identification.

II. Related works

Rosenblum et al. in a series of two seminal papers [17] and [7] 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. While they report excellent precision

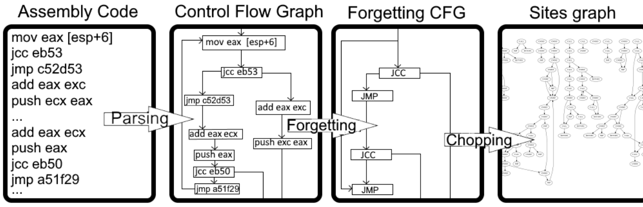

Fig. 1. Overall architecture of the preprocessing phase

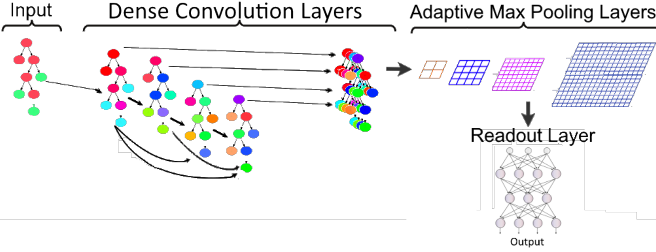

Fig. 2. Overall architecture of a Site Neural Networks

and accuracy, they consider a rather restricted data set in terms of code diversity, compilers families, versions, and options. There are 175 different codes. Only 9 compiler versions are considered and optimization prediction is either 'Low' (e.g. -$O0$) or 'High' (e.g. -$O3$). We show in Section VI-B that more diversity significantly impacts results, hence we carry out our experiments in a more diverse setting.

Rahimian et al. [10] developed BinComp, with a complex model based on three layers – the last one being an Annotated Control Flow Graphs. All these features are embedded into a vector by applying a neighbor hash graph kernel. Some features required the binary codes to be unstripped.

More recently, three papers were published on toolchain provenance. Yang et al. [8] extracts 1024 bits from the object file and process them with a one-dimensional CNN. Chen et al. [9] develop Himalia, which is 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 the prologue and epilogue of functions, and as a result, can 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. [11]. They propose a graph embedding neural network based on methods developed in the field of NLP. The learning phase is composed of two stages. The first stage transforms sequences of instruction in basic blocks using an instruction embedding called i2v and a Recurrent Neural Network. Then, the overall CFG is embedded into a graph, which is aggregated by a 2-round convolution process. Our approach is shown to have a much better learning time for overall better precision and accuracy (Section VI-A).

III. Background

In this paper, we use Graph Neural Networks (GNN) [14], [18], [19] and more precisely Graph Convolutionnal Networks (GCN). The architecture of a GCN is mostly based on a transformation of classical CNN architecture. For example, Zhao et al. [15] generalizes ResNet [20] and DenseNet [21] to graphs. Inputs of a graph neural network is the representation of a graph. The recent survey by Hamilton et al. [22] discusses the different graph representations. 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 a 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 [23] or such as adaptive max pooling by dividing the matrix at each convolution in a fixed number of parts [24], as illustrated in Figure 4.

Note that we name each node with an identifier to increase the predictivity of the model [25].

IV. Our method for toolchain provenance identification

A. Binary code preprocessing

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 I
AUTOMATIC LABELLING

The architecture of the preprocessing is shown in Figure 1. Inputs are binary codes. The Control Flow Graph (CFG) is extracted using a concolic disassembler framework [26]. Then, there are two more steps that we call the *forgetful phase* and the *chopping phase*. They reduce drastically the dimension of the adjacency matrix and facilitate transmission during the convolution phase.

B. The forgetful phase

The forgetful phase consists of 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 I.

```
...
test eax, eax
jz short 424BED
jmp 424D75
mov [ebp+var_1110],1
jmp short 424C08
...
pop     edi
mov     esp, ebp
pop     ebp
retn
```

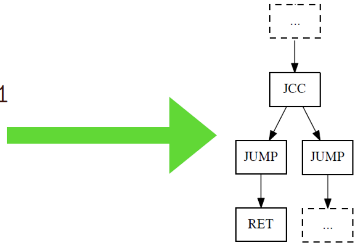

Fig. 3. The forgetful phase

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.

C. 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 α nodes. Sites are obtained using a breadth-first search algorithm from the forgetting graph – the algorithm is presented in Algorithm 1. It performs a limited exploration of the graph using a breadth-first search. During that exploration, it collects from the graph multiple small subgraphs.

We associate with each node of a site two features composed of the instruction type (see Table I) and a unique identifier to solve the problem of anonymity.

Since each site has a small diameter of at most α, 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.

D. Site Neural Networks

Inputs are graphs built from a binary code by the two previous phases. 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 $n \times 2$.

a) Mini-batches: A single input is a set of sites that are collectively regrouped in a graph. In the training phase, the input graphs are partitioned into mini-batches. This allows us to normalize the data [27]. 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 $\text{relu}(x) = \max(0, x)$.

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, α 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 $|N| < \alpha$ **and** $|q1| > 0$ **do**
    empty $q2$
    for all $x \in q1$ **do**
      if $|N| > \alpha$ **then**
        break
      end if
      for all y **such that** $(x, y) \in E$ **do**
        if $|N| > \alpha$ **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

b) Dense 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. Using dense convolutions introduced by Huang et al. [21] the output of one step is fed into every future step.

c) Dimensions: Let d_t be the hyperparameter corresponding to the second dimension of the matrix W_t at convo-

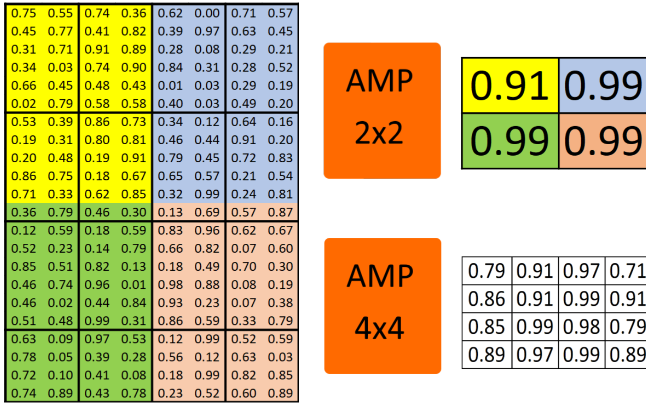

Fig. 4. An example of two adaptive max pooling on a matrix of dimension 22×8. The first adaptive max pooling is of dimension 2×2, it has a kernel size of dimension 11×4. The second adaptive max pooling is of dimension 4×4, it has a kernel size of dimension 6×2.

lution t. The first dimension of the matrix W_k, for $k > 0$, is $\sum_{t=0}^{k-1} d_t$.

d) Output: The dimension of 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.

e) Adaptive Max Pooling Layer(s): Following [16], 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 (AMP) step. This pooling operation is defined by an operator $\mathrm{amp}_{n,m}$ 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 $\left\lceil \frac{u}{x} \right\rceil \times \left\lceil \frac{v}{y} \right\rceil$. We take the maximum in each block. The figure 4 illustrates the adaptive max pooling computation. We iterate four times adaptive max pooling to extract a fixed-size representation of X.

f) 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.

V. Evaluation

A. Dataset

We evaluate the performance of our framework on a data set coming from CodeForces[2].

The Codeforces Dataset: The dataset is made from 36,272 C/C++ source code examples, that solve 91 problems from Codeforces. We compiled them using clang 3.9.1, clang 4.0.1, clang 5.0.1, clang 6.0.0, clang 7.0.0, clang 8.0, gcc 4.8.5, gcc 5.5.0, gcc 6.5.0, gcc 7.5.0, gcc 8.4.0, gcc 9.3.0, mingw 3.4.5, mingw 4.4.1, mingw 4.7.1, mingw 4.9.2, mingw 5.11.0, mingw 8.1.1, visual studio (vs) 10.0, vs 12.0, vs 14.0, vs 2017 and vs 2019. Thus, there are *23 different compilers in total*. The target platform of GCC and Clang binaries is Ubuntu 18.04.4 x64. The target platform of MinGW and Visual Studio binaries is Windows 10 Enterprise x64.

[2] https://codeforces.com/

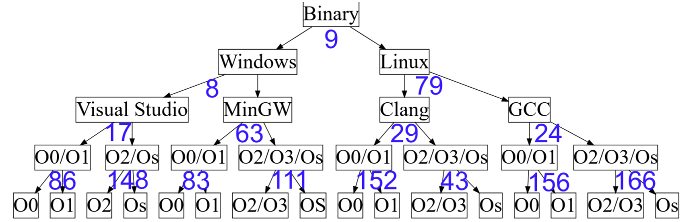

Fig. 5. The hierarchical classifier for optimization level prediction. A parent node is an expert trained during the number of epoch noted in blue.

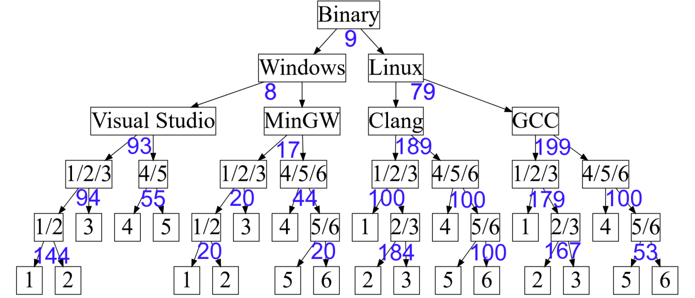

Fig. 6. The hierarchical classifier for version prediction. A parent node is an expert trained during the number of epoch noted in blue.

Compilation process: Each source code is compiled using a random compiler with a random optimization among options -$O0$, -$O1$, -$O2$, -$O3$, and -Os. Binaries are then stripped. We thus have a relatively well balanced dataset of $36,272$ binaries.

Optimization level similarities: There are huge similarities between programs compiled with $-O2$ and -$O3$. Over a random sample of 2920 MinGW binary codes, 561 binary codes compiled with -$O2$ are identical when compiled with -$O3$. This has already been reported [28] and has been an issue for Chen et al. [9] in their compiler optimization detection framework. Note that this is not the case for Visual Studio Compiler which has only one advanced optimization level. Due to the similarity between programs produced by -$O2$ and by -$O3$, we will consider -$O2$ and -$O3$ as a proper subclass.

B. Hierarchies of SNN

Prior works [7]–[9], [11], [17] take each classification task, such as determining the optimization level, separately, and use a single neural network to classify at the level of functions. We choose to implement binaries hierarchical classifiers with a local expert classifier per parent node [29]. In this way, the optimization level prediction depends on the compiler family. Moreover, we take advantage of our domain problem, for instance, we expect -$O0$ instances and -$O1$ instances to be closer to each other than -$O0$ instances and -$O2$/-$O3$ instances.

A site neural network expert is specialized to make the separation between two choices (e.g. between MinGW -$O0$/-$O1$ and MinGW -$O2$/-$O3$). The hierarchy for optimization level prediction (Figure 5) is built upon the prediction of the compiler family. There is one expert per parent node,

thus 15 experts. Similarly, the hierarchy for version prediction (Figure 6) is built upon family prediction and contains 22 experts. Three experts for compiler family prediction are in both hierarchies. Therefore, we have in total 34 experts.

C. Implementation of SNN

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 $\mathrm{amp}_{2,2}$, $\mathrm{amp}_{4,4}$, $\mathrm{amp}_{8,8}$, and $\mathrm{amp}_{16,16}$ operators. The multilayer perceptron has four layers. Their respective number of neurons are 384, 256, 128, and 2. As a result, each of our SNN has $286,962$ trainable parameters. Therefore the complete model, with two hierarchies, has $9,756,708$ trainable parameters. The number of training epochs depends on each expert and is put under each node in Figures 5 and 6. Batch size is 20, learning rate is 0.005 and the loss function is cross-entropy loss.

We implemented our neural network using the language python along with the machine learning library PyTorch. Prepossessing, forgetful, and chopping phases are implemented in C++. ***Data and tools are available at*** https://gitlab.inria.fr/tbenoit/saner-2021-binary-level-toolchain-provenance-identification.

D. Research questions

We investigate the following research questions, ending the two most crucial ones, to validate our framework and to see its current limits:

RQ1 How does our framework evolve when the site size α increases in term of running time performance?

RQ2 How does our framework evolve when the site size α increases in term of accuracy?

RQ3 Does our framework have the capacity to predict the compiler and optimization level of binary codes?

RQ4 Does our framework have the capacity to predict the compiler version of binary codes?

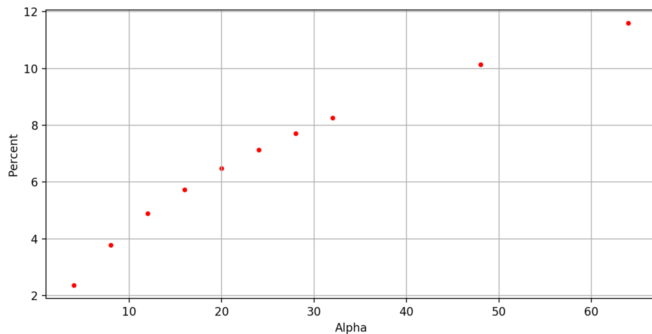

Fig. 7. Percent of the CFG dataset in megabyte kept by the chopping phase.

E. RQ1: How does our framework evolve when the site size α increases in terms of running time performance ?

a) Goal: The whole control flow graph has variable size and is not suitable as direct input of a machine-learning process. As a result, we chop the control flow graph in 100 smaller graphs of fixed size α. The dimension of the parameter α modifies the processing time necessary to perform the chopping phase itself along with the learning phase. We want to evaluate the impact of this parameter α in the learning phase process of both our hierarchies the one for the optimization level prediction and the one for the compiler version prediction.

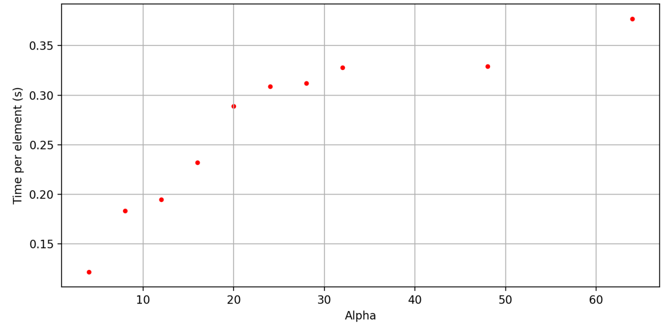

Fig. 8. Chopping phase: average time to process a binary code

α	Option s/e	Version s/e	Complete s/e
4	0.46	0.58	0.95
8	0.59	0.74	1.21
12	0.73	0.80	1.39
16	0.71	0.85	1.42
20	0.85	0.97	1.67
24	0.79	0.90	1.54
28	0.84	0.93	1.61
32	0.85	1.02	1.71
48	0.98	1.08	1.88
64	1.13	1.25	2.20

TABLE II
LEARNING PHASE AVERAGE TIME (IN SECONDS) PROCESSING AN BINARY

b) Method: We vary α by taking values : 4, 8, 12, 16, 20, 24, 28, 32, 48 and 64. As shown in Figure 7, using those α in the chopping phase retains from 2% of the original CFG data to 12%. This is a good range of values as we seek to retain only a small part of the forgetting graphs data.

We select only a sample of 1000 binaries from our dataset. Using an identical computer sequentially for all values of alpha, we learn one epoch for each expert. We measure the learning phase processing time in function of α. We extrapolate from that the average learning phase processing time per binary if the number of epochs was correct. We also record the average processing time of the chopping phase of a binary in function of α. We present the average time in seconds of the process a binary coming from our dataset.

We run experiments on a computer equipped with an Intel i7-8665U, 4 cores, and a frequency of 2.11 GHz.

c) Results: Graph extraction, which does not depend on α, takes an average processing time of 0.11 seconds per binary. The average time of the chopping phase goes from 0.12 seconds per binary to 0.38 seconds per binary depending

on α. Chopping processing time is monotonic with α. The average time of the complete learning phase processing goes from 0.95 seconds per binary to 2.21 seconds per binary.

d) Conclusion: As α increases, so do the volume of the data to be classified. Thus one can adapt the process to its need using α. It can scale to dozens of thousands of binaries. We note that extraction and chopping phases can be parallelized efficiently. Due to the use of experts, the learning phase is also done in parallel.

F. RQ2: How does our framework evolve when the site size α increases in terms of accuracy ?

a) Goal: We want to identify the impact of α on the accuracy of our framework. Indeed, since the parameter α gives the size of data extracted, at first glance, it must affect the accuracy of the machine learning process. However, it is a bit tricky because machine learning neural networks have an inherent variance in their results. To measure the performance of our hierarchies, we use as a single metric the macro average F1-Score. It is a simple metric that is unbiased by potential class imbalances.

b) Methodology: As in RQ1, we apply our framework with 4, 8, 12, 16, 20, 24, 28, 32, 48, and 64 as different possible values of α. We effectuate ten runs for each alpha. In each run, we sample 10% of our dataset as the train set. We sample a test set of size 2000. A different validation set composing 10% of the train set is used for each expert. For each specialized site neural network, we select the model neural network with the best accuracy on the validation split on the last epoch. We train our hierarchies for the optimization level prediction task and the compiler version prediction task.

To deal with the inherent variance, we use the mean value of each 10 runs as the estimated macro average F1-Score for a given α. Our data is then analyzed using linear models with the ordinary least square (OLS) method. This simple model assumes that our data follow a linear law with some noise that should be identically distributed. We evaluate the impact of α on the macro average F1-Score by the r^2 value of the statistical analysis. We can also give a probability that the model fits the data using the F1-Statistic.

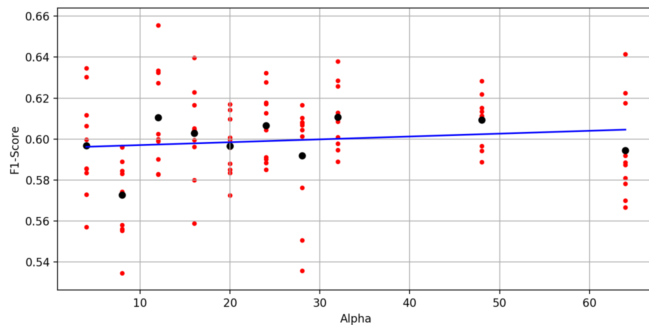

Fig. 9. Macro average F1-Score along alpha on the optimization level prediction task. Red dots are values obtained by a run. Black dots are mean values for an α. The blue line is a linear model using the ordinary least square method (OLS).

c) Results: On the optimization prediction task, a linear model does not fit the data using the ordinary least square (OLS) method (Figure 9). First, the probability that there is no relationship is too high as demonstrated by a p-value of 0.537. Even if there were a relation, α would explain less than 5 percent of the variance in the overall macro average F1-Score.

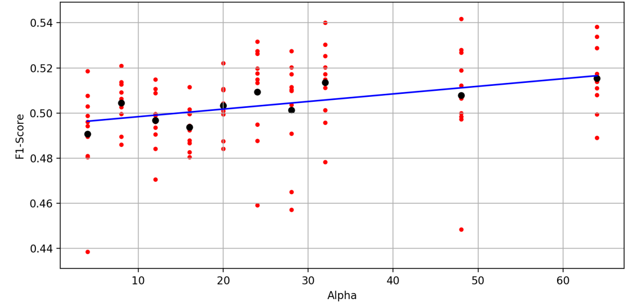

Fig. 10. Macro average F1-Score along alpha on the compiler version prediction task. Red dots are values obtained by a run. Black dots are mean values for an α. The blue line is a linear model using the ordinary least square method (OLS).

On the version prediction task, a linear model is a better fit for the data using the ordinary least square (OLS) method.

First, the p-value is 0.00991. This probability may be lower as indicated by the Durbin-Watson test result of 2.699. Such a value higher than 2 predicates negative auto-correlation which could artificially increase the p-value by breaking an assumption of the ordinary least squares method. However, the probability of the Omnibus test about the distribution of errors is 0.375 which is too close to 0. With a good probability, the model fails to have normally distributed errors, a potential break of another assumption. The r^2 of the model is an intermediate value of 0.585, α could explain around 58% of the variation in the mean F1-Score.

d) Conclusion: We have moderate confidence that increasing alpha does increase our capacity to predict the data in the version prediction task. We select $\alpha = 32$ as the value to use in our next questions due to it having achieved the best mean F1-Score in both prediction tasks and extracting only 8% of the complete CFG data in megabytes.

G. RQ3: Does our framework have the capacity to predict the compiler and optimization level of binary codes?

a) Methodology: We set the site size to be $\alpha = 32$ (cf. RQ1-RQ2). We train experts of the first hierarchies to predict both the compiler family and the optimization level. To run this experiment, the dataset is split into a train set and a test set. The test set has a size of 2200 and is balanced along the dimension of the combination of compilers, compiler versions, and optimization levels. A different validation set composing 10% of the train set is used for each expert. The loss function is the cross entropy loss. For each specialized site neural network, we select the epoch with the best accuracy on the validation split on the last ten epochs. After training, the hierarchical classifier is evaluated thanks to the test set.

		Clang				GCC				VS				MinGW			
		O0	O1	O2/O3	Os	O0	O1	O2/O3	Os	O0	O1	O2	Os	O0	O1	O2/O3	Os
Clang	O0	95.83	5.83	0.83	1.67	0	0	0	0	0	0	0	0	0	0	0	0
	O1	1.67	55	4.58	2.5	0	0	0	0	0	0	0	0	0	0	0	0
	O2/O3	1.67	29.17	79.17	53.33	0	0	0	0	0	0	0	0	0	0	0	0
	Os	0.83	8.33	15	41.67	0	0	0	0	0	0	0	0	0	0	0	0
GCC	O0	0	0	0	0	98.33	6.67	0	0.83	0	0	0	0	0	0	0	0
	O1	0	0	0	0	1.67	92.5	2.5	0.83	0	0	0	0	0	0	0	0
	O2/O3	0	0	0.42	0	0	0.83	89.58	9.17	0	0	0	0	0	0	0	0
	Os	0	0	0	0.83	0	0	7.92	89.17	0	0	0	0	0	0	0	0
VS	O0	0	0	0	0	0	0	0	0	46	4	5	52	0.83	0	0.42	0.83
	O1	0	0.83	0	0	0	0	0	0	8	84	6	6	0	0	0	0.83
	O2	0	0	0	0	0	0	0	0	4	10	85	1	0	0.83	0	0
	Os	0	0.83	0	0	0	0	0	0	42	2	3	41	0	0	0	0
MinGW	O0	0	0	0	0	0	0	0	0	0	0	0	0	83.33	9.17	1.25	2.5
	O1	0	0	0	0	0	0	0	0	0	0	0	0	5.83	68.33	4.17	6.7
	O2/O3	0	0	0	0	0	0	0	0	0	0	1	0	5	16.67	79.58	25
	Os	0	0	0	0	0	0	0	0	0	0	0	0	5	5	14.58	64.17
		100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100

Fig. 11. Confusion matrix of SNN on the compiler family and option prediction. On the diagonal, the best value is 100 and elsewhere it is 0.

Compiler	Precision	Recall	F1 Score	Support
Clang	1	0.9933	0.9967	600
GCC	0.9967	1	0.9983	600
MinGW	0.9983	0.9917	0.9950	600
VS	0.9828	0.9975	0.9901	400
Macro AVG	0.9944	0.9956	0.9950	2200

TABLE III
F1-SCORE OF SNN FOR COMPILER FAMILY PREDICTION.

Option	Precision	Recall	F1 Score	Support
-O0	0.7917	0.8261	0.8085	460
-O1	0.8289	0.7478	0.7863	460
-O2/-03	0.7869	0.8329	0.8092	820
-Os	0.6316	0.6	0.6154	460
Macro AVG	0.7598	0.7517	0.7549	2200

TABLE IV
F1-SCORE OF SNN FOR OPTIMIZATION LEVEL PREDICTION.

b) Results: Table III presents F1-Score for each compiler family and the macro average F1-Score. We achieve a very high overall F1-Score of 0.9950 on the compiler family prediction. Table IV presents F1-Score for each optimization level and the macro average F1-Score. We achieve a high F1-Score of 0.7549 on the optimization level prediction.

c) Conclusion: We achieve a very high F1-Score on the compiler family prediction. It is so high that nearly all errors are in predicting the optimization level. On this other task, we achieve a high F1-Score. We conclude that predicting the optimization level is harder than predicting the compiler family. We suppose this is in part due to the similarity of binaries produced by different compiler options.

Version	Precision	Recall	F1-Score	Support
Clang 3.9.1	0.3137	0.16	0.2119	100
Clang 4.0.1	0.194	0.26	0.2222	100
Clang 5.0.1	0.1897	0.11	0.1392	100
Clang 6.0.0	0.1594	0.11	0.1302	100
Clang 7.0.0	0.216	0.27	0.24	100
Clang 8.0.0	0.1384	0.22	0.1699	100
GCC 4.8.5	0.9375	0.90	0.9184	100
GCC 5.5.0	0.6032	0.38	0.4663	100
GCC 6.5.0	0.4237	0.75	0.5415	100
GCC 7.5.0	0.6747	0.56	0.612	100
GCC 8.4.0	0.3636	0.24	0.2892	100
GCC 9.3.0	0.4359	0.51	0.47	100
MinGW 3.4.5	1	0.99	0.995	100
MinGW 4.4.1	0.99	0.99	0.99	100
MinGW 4.7.1	0.9794	0.95	0.9645	100
MinGW 4.9.2	0.95	0.95	0.95	100
MinGW 5.11.0	0.98	0.98	0.98	100
MinGW 8.1.1	1	1	1	100
VS 10.0	0.939	0.9625	0.9506	80
VS 12.0	0.8478	0.975	0.907	80
VS 14.0	0.9125	0.9125	0.9125	80
VS 2017	0.9367	0.925	0.9308	80
VS 2019	0.9452	0.8625	0.902	80
Macro AVG	0.6578	0.6508	0.6475	2200

TABLE V
F1-SCORE OF SNN FOR VERSION PREDICTION

H. RQ4: Does our framework have the capacity to predict the compiler version of binary codes?

a) Methodology: We set the site size to be $\alpha = 32$ (cf. RQ1-RQ2). We use the same methodology as before with experts of the second hierarchy to predict the compiler version.

		Clang						GCC						MinGW						VS				
		3.9	4.0	5.0	6.0	7.0	8.0	4.8	5.5	6.5	7.5	8.4	9.3	3.4	4.4	4.7	4.9	5.11	8.1	10.0	12.0	14.0	2017	2019
Clang	3.9	16	8	10	7	4	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	4.0.	25	26	19	20	20	24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	5.0	10	10	11	6	12	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	6.0	11	9	17	11	5	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	7.0	18	14	16	27	27	23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	8.0	20	31	26	28	32	22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
GCC	4.8	0	0	0	0	0	0	90	1	3	0	1	1	0	0	0	0	0	0	0	0	0	0	0
	5.5	0	0	0	0	0	0	2	38	15	3	3	2	0	0	0	0	0	0	0	0	0	0	0
	6.5	0	1	0	0	0	0	4	54	75	18	16	9	0	0	0	0	0	0	0	0	0	0	0
	7.5	0	0	0	0	0	0	1	2	4	56	8	12	0	0	0	0	0	0	0	0	0	0	0
	8.4	0	0	0	0	0	0	2	2	2	11	24	25	0	0	0	0	0	0	0	0	0	0	0
	9.3	0	0	1	0	0	0	1	3	1	12	48	51	0	0	0	0	0	0	0	0	0	0	0
MinGW	3.4	0	0	0	0	0	0	0	0	0	0	0	0	99	0	0	0	0	0	0	0	0	0	0
	4.4	0	0	0	0	0	0	0	0	0	0	0	0	0	99	0	0	0	0	0	0	1.25	0	0
	4.7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	95	2	0	0	0	0	0	0	0
	4.9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	95	0	0	0	0	0	0	0
	5.11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	98	0	0	0	0	0	0
	8.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	100	0	0	0	0	0
VS	10.0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	96.25	1.25	2.5	1.25	0
	12.0	0	1	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	2.5	97.50	3.75	0	6.25
	14.0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	1.25	91.25	2.5	2.5
	2017	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.25	92.5	5
	2019	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.25	0	0	3.75	86.25
		100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100

Fig. 12. Confusion matrix of SNN on the compiler version prediction. On the diagonal, the best value is 100 and elsewhere it is 0.

Family	Precision	Recall	F1-Score	Support
Clang	0.2019	0.1883	0.1856	600
GCC	0.5731	0.5567	0.5496	600
MinGW	0.9832	0.9767	0.9799	600
VS	0.9162	0.9275	0.9206	400
GCC-MinGW-VS	0.8242	0.8202	0.8167	1600
MinGW-VS	0.9497	0.9521	0.9502	1000

TABLE VI
F1-SCORE OF SNN FOR VERSION PREDICTION AMONG COMPILERS FAMILY

b) Results: Table V presents F1-Score for each compiler version, and macro averages F1-Score. We achieve a moderate F1-Score of 0.6475 on the version prediction task. Table VI presents macro averages F1-Score of compiler version identification by compiler family. We notice the macro average F1-Score of 0.1856 for the Clang family. The Clang family version prediction stands out due to this very low accuracy. On the other hand, the macro average F1-Score of the MinGW family is about 0.9799. Table VI shows that if we omit Clang, we attain a macro average F1-Score if 0.8167. Moreover, if we concentrate on binaries compiled for Windows, we attain a very high F1-Score of 0.9502.

c) Conclusion: We conclude that compiler version prediction is hard on the Clang family but easy for binaries coming from the Windows platform.

VI. Discussion

We choose to compare our framework to the work of Massarelli et al. [11]. As we said previously, it is one of the closest frameworks to ours. Indeed, they used graph convolution at the binary function level while we performed graph convolution at the binary level. Moreover, this framework is available in a repository. This is not the case with other works [7]–[10] which makes it difficult for a fair comparison. Still, we also provide elements of comparison with Rosenblum et al. [17].

A. Comparison with Massarelli et al. [11]

a) Methodology: Massarelli et al. [11] relies on radare2 [30] to extract CFG of binary functions in a binary. They propose an instruction embedding method called i2v to represent each instruction, and then feed it to an RNN.

We sample a set of 2843 binaries from our dataset that we transform to CFG of functions using the Massarelli et al. framework. Using Massarelli et al. framework, 227 of these binaries constitute the validation split, 601 constitute the test split and the rest belong to the train split. *We take a much smaller dataset that for RQ3 and RQ4 because Massarelli et al. preprocessing phase is around 1300 time slower than ours for learning.*

On our side, we train our framework using a dataset containing the same 2843 binaries with a random test set of 601 binaries. We perform the experiment 10 times to mitigate randomness.

To compare with our binary level approach, we take two distinct approaches:

- **Function-level** We evaluate Massarelli et al., denoted MA. We transform the output of our SNN framework from binary level to function level. To do so, the prediction of each function is the prediction for the binary containing the function. We use the information of MA to get the number of functions in each binary. We note this process SNN-F;
- **Program-level** We already have evaluated our approach with a complete dataset. We note this evaluation SNN-Full. We transform the output of MA from function level to binary level. To do so, the prediction for a given binary

Framework / Task	Compiler	Optimization	Version
MA-B	0.91	0.42	0.32
SNN	0.92 [$\pm$0.03]	0.58 [$\pm$0.03]	0.45 [$\pm$0.02]
SNN-Full	**0.99**	**0.75**	**0.65**
MA	**0.90**	0.36	0.36
SNN-F	0.87 [$\pm$0.07]	**0.60** [$\pm$0.05]	**0.42** [$\pm$0.02]

TABLE VII
MACRO AVG F1-SCORE OF DIFFERENTS FRAMEWORK WITH EACH PREDICTION TASKS.

is the majority prediction of each function extracted from the binary. We note this process MA-B;

b) Results: Massarelli et al. preprocessing is complicated and time-consuming, as radare2 disassembly is a costly process. The average time per binaries is 665 seconds with a minority of binaries taking hours to complete. While this preprocessing is done in parallel, it takes *525 hours* of computation to obtain 2842 binaries. It was done on computers equipped with two Intel Xeon silver 4110. The dataset for Massarelli et al. approach contains approximately 825,000 binary functions. Each epoch takes approximately 2200 seconds. Thus, the average learning time is 116 seconds per binary, using computers equipped with two Intel Xeon gold 5218R – while on an Intel i7-8665U our approach has an average learning time of 1.71 seconds per binary.

We report the result of each framework on the three tasks in Table VII. For the sake of brevity, we report only the overall macro average F1-Score with two digits along with standard deviation when available. Comparing SNN with a restricted dataset to MA-B, we are on the same range on family prediction. However, we outperform MA-B by 0.16 on optimization prediction, and 0.13 on version prediction. Using our full dataset, SNN-Full outperforms MA-B by 0.08 en compiler family prediction, 0.33 on optimization prediction and 0.33 on version prediction. This is expected as MA-B had less training data than SNN-Full. On the other side, comparing MA to SNN-F, we are again on the same range on family prediction. We outperform it by 0.24 on optimization prediction and 0.06 on version prediction.

c) Conclusion: We are approximately 68$\times$ times faster during learning, and 1300$\times$ time faster during preprocessing. Moreover, we can use much more parallelism in the learning phase. This processing time allows us to enhance our classification performance by learning from more binaries.We are also able to consistently outperform in terms of accuracy MA except for the compiler family where we are in the same range.

B. Comparison with Rosenblum et al. framework [17]

Rosenblum et al. [17] dataset and classifier are not available, so we give only elements of comparison. They report accuracy of 0.999 for compiler family, 0.999 for optimization level, and 0.918 for compiler version. However, their dataset consists of 2,686 programs compiled from *only 175 different source code examples*, while we consider 36,272 different source code examples (internal validity is ensured by random assignment of a code to a toolchain). They consider three compiler families (Visual Studio, Intel Compiler, and GCC), but only 9 compiler versions, and the optimization prediction is reduced to 'Low' (e.g. $-O0$) vs 'High' (e.g. $-O2$) – hence a total of 18 compiler configurations where we consider 92 such configurations.

We can observe trends when we move from our broad dataset to a more restricted one closer to Rosenblum et al. setting. We select compiler versions VS 10.0, VS 12.0, VS 2017, GCC 4.8, GCC 5.5 and GCC 7.5. Reproducing the setting of Rosenblum et al., we restrict optimization option prediction to a choice between 'Low' and 'High' optimization options. After a learning phase on this restricted dataset, our F1-Scores is 1.0 for compiler family, 0.97 for optimization level and 0.89 for compiler version. *As expected, broadness of the dataset significantly impacts accuracy.*

VII. LIMITATIONS

Our dataset is composed of small programs (most file sizes are around 30kb). It would be interesting to use a more diverse dataset. Nevertheless, our evaluation at least demonstrates the accuracy of SNN. Moreover, our approach was tested and validated on 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.

VIII. CONCLUSION

We consider the problem of toolchain provenance identification. Our starting hypothesis is that binary code is not unstructured data and that semantics is important. Moreover, since libraries are more frequently dynamically linked, binaries are usually homogeneous in terms of toolchain provenance.

In this work, we explore the possibility of (i) extracting semantic features in the form of graphs, (ii) processing the neural networks of the graphs to propagate the information according to the topology of the graph, and (iii) using tailored hierarchies to fit a dataset. We demonstrate that binary-level toolchain provenance identification is feasible with both high precision/accuracy and fast learning – we outperform a recent function-level approach on these metrics.

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. Also, the pooling layers play an important role. It should be worth looking at which features are useful and how they are intertwined to improve pooling. This question bounces off the question of semantics. Finally, CFG provides only a very shallow program semantics. An interesting question would be to automatically extract and take advantage of some sort of richer semantic features without too much extra cost.

ACKNOWLEDGMENTS

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

References

[1] M. J. Hohnka, J. A. Miller, K. M. Dacumos, T. J. Fritton, J. D. Erdley, and L. N. Long, “Evaluation of compiler-induced vulnerabilities,” *Journal of Aerospace Information Systems*, vol. 16, no. 10, pp. 409–426, 2019.

[2] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, “Deep learning code fragments for code clone detection,” in *2016 31st IEEE/ACM ASE*, 2016, pp. 87–98.

[3] J. Calvet, J. M. Fernandez, and J.-Y. Marion, “Aligot: Cryptographic function identification in obfuscated binary programs,” in *Proceedings of the 2012 ACM CCS*, 2012, p. 169–182. [Online]. Available: https://doi.org/10.1145/2382196.2382217

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

[5] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, “BYTEWEIGHT: Learning to recognize functions in binary code,” in *23rd USENIX Security Symposium*, 2014, pp. 845–860. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/bao

[6] P. Shirani, L. Wang, and M. Debbabi, “Binshape: Scalable and robust binary library function identification using function shape,” in *DIMVA*. Springer Publishing, 2017, pp. 301–324.

[7] N. Rosenblum, B. P. Miller, and X. Zhu, “Recovering the toolchain provenance of binary code,” in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011, p. 100–110. [Online]. Available: https://doi.org/10.1145/2001420.2001433

[8] 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*, 2019, pp. 1–6.

[9] 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*. Springer Publishing, 2019, pp. 35–47.

[10] A. Rahimian, P. Shirani, S. Alrbaee, L. Wang, and M. Debbabi, “Bincomp: A stratified approach to compiler provenance attribution,” *Digital Investigation*, vol. 14, pp. S146 – S155, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1742287615000602

[11] L. Massarelli, G. Luna, F. Petroni, and L. Querzoni, “Investigating graph embedding neural networks with unsupervised features extraction for binary analysis,” *Workshop on Binary Analysis Research*, 2019.

[12] A. Viet Phan, M. Le Nguyen, and L. Thu Bui, “Convolutional neural networks over control flow graphs for software defect prediction,” in *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, 2017, pp. 45–52.

[13] K. Redmond, L. Luo, and Q. Zeng, “A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis,” *CoRR*, vol. abs/1812.09652, 2018.

[14] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, “Graph neural networks: A review of methods and applications,” 2018.

[15] W. Zhao, C. Xu, Z. Cui, T. Zhang, J. Jiang, Z. Zhang, and J. Yang, “When work matters: Transforming classical network structures to graph cnn,” *arXiv:1807.02653*, 2018.

[16] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, “Weisfeiler-lehman graph kernels,” *Journal of Machine Learning Research*, vol. 12, no. 77, pp. 2539–2561, 2011. [Online]. Available: http://jmlr.org/papers/v12/shervashidze11a.html

[17] N. Rosenblum, B. Miller, and X. Zhu, “Extracting compiler provenance from program binaries,” *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp. 21–28, 2010.

[18] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, “The graph neural network model,” *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2008, publisher: IEEE.

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

[20] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” 2015.

[21] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, “Densely connected convolutional networks,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[22] W. L. Hamilton, R. Ying, and J. Leskovec, “Representation learning on graphs: Methods and applications,” *arXiv preprint arXiv:1709.05584*, 2017.

[23] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, “An end-to-end deep learning architecture for graph classification,” in *32th AAAI Conference on Artificial Intelligence*, 2018.

[24] J. Yan, G. Yan, and D. Jin, “Classifying malware represented as control flow graphs using deep graph convolutional neural network,” in *49th Annual IEEE/IFIP DSN*, 2019, pp. 52–63.

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

[26] G. Bonfante, J. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry, “CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions,” in *22nd ACM Conference on Computer and Communications Security*, Denver, United States, Oct. 2015. [Online]. Available: https://hal.inria.fr/hal-01257908

[27] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” 2015.

[28] M. Egele, M. Woo, P. Chapman, and D. Brumley, “Blanket execution: Dynamic similarity testing for program binaries and components,” in *23rd USENIX Security Symposium*, 2014, pp. 303–317.

[29] C. Silla and A. Freitas, “A survey of hierarchical classification across different application domains,” *Data Mining and Knowledge Discovery*, vol. 22, pp. 31–72, 01 2011.

[30] R. Team, “Radare2 book,” in *GitHub*, 2017.