BinSeg: Leveraging Semantic Segmentation for Code and Data Separation in Non-standard Binary Formats

Hadjer Benkraouda[1], Aizaz Ansari[2⋆], Deebthik Ravi[3⋆], Gautham Dinesh[2], and Michail Maniatakos[1]

[1] Center for Cyber Security, New York University Abu Dhabi, UAE
[2] Electrical and Computer Engineering, New York University Abu Dhabi, UAE
[3] National Institute of Technology, Tiruchirappalli, India

Abstract. Static binary analysis has been a very integral step in software security analyses such as reverse engineering and vulnerability/malware discovery. This method is especially useful when the source code of the software under inspection is proprietary or unavailable. A crucial step to ensure robust static analysis is differentiating code from data. Correctly marking the boundaries between data and code sections avoids errors due to misinterpreting data as executable instructions and vice-versa. Classical binary analysis tools such as IDA Pro and Ghidra have been very successful with known binary formats i.e. PE, ELF, and Mach-O. However, one of the major limitations of the current binary analysis tools is their inability to automatically analyze atypical formats. Due to the advent of Industry 4.0 and the proliferation of IoT devices, an increase of binary files of unknown formats has surfaced. Manually reverse engineering the details of each format has a prohibitive time cost. Therefore, an important and urgent challenge for securing these devices is to automate the binary analysis for unknown binary formats. In this paper we aim to automate one of the stages of binary analysis, code and data separation. Our method converts the binary files into images. Next, our method leverages image-based machine learning algorithms to perform semantic segmentation to identify the data and code sections of binary images. We train our model on known binary formats (ELF) and then test it on binary files of non-standard formats (programmable logic controller binaries). Our results show that the trained model has achieved a maximum accuracy value ranging between 91.91% and 89.22% depending on the underlying methodology used for interpreting binary files. The results also show that our suggested method performs very well for code section detection, while it under-performs for embedded bytes.

Keywords: Binary analysis, Machine learning, Industrial control systems security

[⋆] Both authors contributed equally to this research

1 Introduction

Static binary analysis is the technique used to analyze programs *sans*-source code without running them. This technique is used indispensably in many areas including performance modeling [18], vulnerability detection [8], binary reverse engineering and reconstruction, binary modification and rewriting when the source code is not available, and static binary translators [5]. Research interest in binary analysis has significantly increased in the last years. Recent projects concerning binary analysis include tools and methods that aid human analysts or automate analysis tasks. They range from disassemblers and decompilers, to complex analysis frameworks [22, 4] that combine static analysis with other techniques, primarily symbolic execution [20, 21], fuzzing [7, 9], or both [23].

To be able to perform static analysis, a tool has to have the ability to disassemble a binary and reconstruct its control flow graph. Modern disassemblers rely heavily on heuristics, this makes the process of disassembly undecidable [26]. The most important step to ensure accurate disassembly is to precisely identify code instructions and data within a binary file. Identifying code instructions/sections is especially hard while performing static analysis because the program can not be run. Although dynamic analysis tools might help in identifying some of the instructions used in the program, they do not generate a complete set of all the instructions within a binary. Additionally, running the binary under test can be dangerous if the binary is malicious or faulty. Standard file formats such as ELF, PE, and Mach-O are organized into sections (e.g., .text, .data) and have headers that list the sections and their relative addresses, including their class (e.g., code, data). Even with addition of metadata that demarcates labeled sections, data bytes can be sometimes included within these sections [4].This happens often through compiler optimizations. Often, compilers pad functions with data bytes to ensure memory alignment or enhanced performance [26]. Embedding data in code sections has also been used in creating adversarial examples for malware samples to evade malware detection tools [13]. For standard formats it is important to identify those embedded data bytes within code sections.

Binary analysis tools and research have been facing a new challenge in the recent years. With the recent spread of IoT devices, many non-standard binary formats have emerged (e.g., PLC binaries [12]). These can be different for each device and vary from one vendor to another. Manually reverse engineering the format of each of these devices can have a prohibitive time cost. Additionally, most embedded device firmware images are not structured into sections, but rather come in the form of a binary blob. For both of the aforementioned binary files (i.e., non-standard and binary blobs) it is important to be able to identify code sections and any data embedded within these sections. In this paper we focus on the problem of identifying code sections and the embedded data bytes within code sections for non-standard binary formats.

[4] We refer to all instructions as code and to everything else as data. We also refer to sections composed of code as code sections and all other sections as data sections.

Recent research efforts have shown that binary analysis can be greatly improved using data visualization techniques. Fields, such as malware detection, have seen significant improvements in accuracy, time, and memory consumption using visualized binaries when compared to raw binary analysis. Visualization based binary analysis methods have been successful due to their ability to leverage the advances in deep learning approaches in recognizing patterns and underlying characteristics of the binary [3]. Motivated by these results, in this work we propose a method that leverages data visualization techniques to create images that correspond to each binary file. Next, our method makes use of semantic segmentation to separate code and data sections in non-standard binary formats. As a results our contribution is 2-fold:

1. We propose a methodology that automates code and data separation for non-standard binary formats by training semantic segmentation models without relying on any ISA specific features.
2. We evaluate our methodology on non-standard binary file formats from real Wago PLCs and show that it presents a viable solution for section-level code and data separation.
3. We created datasets for both standard and non-standard binary formats with their corresponding images and ground-truth class labels.

2 Preliminaries

2.1 Semantic Segmentation

Semantic segmentation is a method developed for object detection within an image. Semantic segmentation leverages neural networks (e.g., DCNNs, CNNs) to correctly classify each pixel within an image to a class. Semantic segmentation can perform *binary* classifications, i.e., foreground and background and it can also perform multi-label classifications (e.g., car, tree, person). Semantic segmentation has been deployed in many critical fields ranging from autonomous vehicles, to human-computer interaction, biomedical image processing, and social media applications. There have been many proposed underlying architectures for semantic segmenting, most prominently, using ”fully convolutional” networks [15], U-Net which was specifically developed for biomedical image processing. This method maintains high accuracy while using a limited number of images for training [17], and DeepLab (the state-of-the-art) which rectifies the loss of localization due to the max-pooling and downsampling introduced by DCNNs by using Conditional Random Field (CRF) [6]. In our implementation, we use Deeplab to train our custom semantic segmentation models.

2.2 Code separation problem

The code separation problem can be divided into 2 sub-problems. The first problem is code section discovery problem. Given a binary b, we are trying to find all the pairs (a_n,b_n) and (c_m,d_m). Here a is the set of all starting addresses of code sections, b is the set of all ending addresses of code sections, n is the number of code sections within the binary, where c is the set of all starting addresses of data sections, d is the set of all ending addresses of data sections, and m is the number of code sections within the binary. For standard binary file formats this information can be found using headers or symbols/metadata added to the binary, while for non-standard binary file formats this problem is more challenging. The second problem is the code instruction discovery problem (known simply as the code discovery problem). After identifying the code and data sections within a binary, this problem becomes: given a section s formed by (a_n,b_n) and (c_m,d_m), we try to locate the bytes that do not belong to the class of the section (i.e., code bytes within data sections and data bytes within code sections). As aforementioned, using a neural network based semantic segmentation model to separate code and data sections and to locate outlier bytes within a section for non-standard binary formats is particularly appealing, since it can be quickly retrained to adapt to a given ISA when no ground-truth data is provided for the non-standard binary format is available.

2.3 Thumb mode

In the case of architectures that support variable-length instructions, when a piece of data is misinterpreted, the following bytes become misaligned, and may lead to erroneous and problematic disassembly. On the other hand, RISC architectures have fixed length instructions, even when some data bytes are misinterpreted as instructions, they rarely result in executable instructions. With the propagation of embedded systems, RISC processors such as ARM and MIPS have become widely used. RISC processors used to have fixed instruction length, but due to the demand for lower memory consumption, they started offering mixed-length instructions (16 bits and 32 bits). In ARM the 16 bit instruction, known as Thumb instruction set, helps in reducing executables' sizes. ARM executable can be built as either ARM-only (32 bit instructions only) or as a mixture of ARM/Thumb instructions (32 and 16 bit instructions). Although mixed executables have instructions of different lengths, they differ from variable-length CISC executables. In ARM/Thumb executables the Thumb instructions and the ARM instructions are not intermingled, instead they are separate regions and use instructions to switch from one mode to the other [5]. While training our model, we used a dataset that included both ARM/Thumb and ARM executables. We included ARM/Thumb executables to ensure that the semantic segmentation models recognize Thumb instructions as code bytes.

2.4 Programmable Logic Controller

PLCs are real-time embedded devices that execute control algorithms to run physical processes [2]. A typical PLC leverages applications that facilitate programming the PLC to perform different tasks. The International Electrotechnical Commission (IEC) defines industry standards for all PLCs in IEC 61131. IEC

61131-3 outlines the software architecture and programming of PLCs by defining programming languages, data types and variable attribution. PLCs use both textual and graphical programming languages, such as ladder diagrams, structured text, instruction list, function block diagrams, and sequential function charts [24].

PLC binary format PLC binaries do not conform to any of the standard binary formats. Still, it has some similarities with standard binary formats. ICSREF [12] reverse engineers the PLC binary format produced using the Codesys compiler. PLC binaries are composed of a header, a section that contains the main program, data sections, and linked libraries. The file includes the header section in the beginning. The header includes the program's entry point, stack size, and dynamic library locator. Unlike conventional headers, this header does not include information about the section offset addresses or classes. Next is a subroutine that sets constant variables and initializes functions. The debugger handler subroutine is located next, this enables dynamic debugging from the IDE. These two subroutines are followed by user-defined and library functions and the main PLC program. Finally, the binary file is ended with data sections.

3 Image based code and data separation

In this section we delve into the details of our proposed methodology for code and data separation. As aforementioned, our method uses data visualization on standard binary files to generate their corresponding images. We also use conventional binary analysis tools to establish the ground truth annotation of code and data bytes within the visualized binaries. Next, the generated dataset of binary image and ground truth labels is used to train DeepLab, a state of the art semantic segmentation model. Fig. 1 gives a visual overview of our proposed methodology.

3.1 Dataset generation

An essential part of designing our solution is building a dataset. Although there are many datasets that collect binary files, both benign and malicious, to the best of our efforts we were not able to find a dataset with labeled bytes. To this end we generated a datasets of visualised binaries and their corresponding labeled bytes. This section describes the dataset and describes the details of the processes.

Raw binary dataset We used a dataset made public by Clemens [1]. The dataset is comprised of benign software and includes binaries from different Instruction Set Architectures (ISA) and standard bianry formats. In total the dataset includes 15,291 executable files for 20 architectures. In our training and validation stages we focused on the 32-bit ARM binaries from this dataset. We

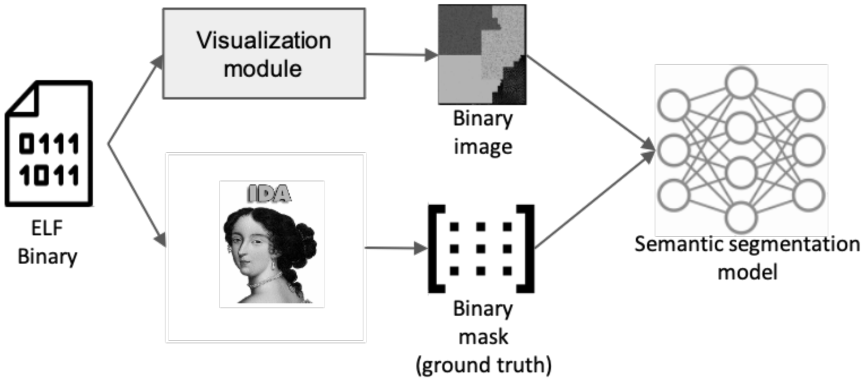


Fig. 1: Overview of the semantic segmentation training Setup.

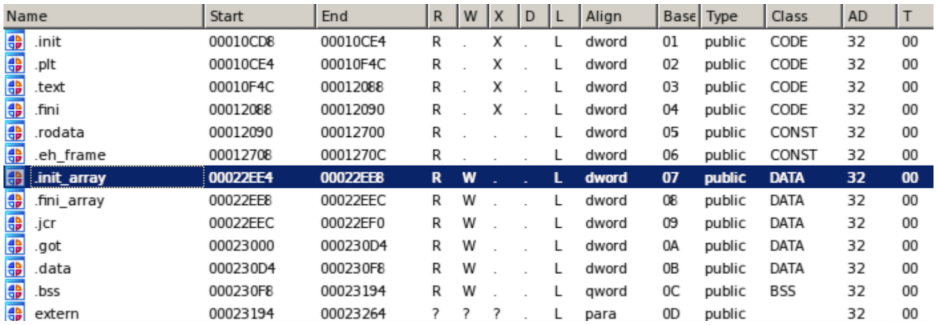
used ARM binaries since the architecture for the non-standard binary format used in our case-study is also 32-bit ARM based. The subset of the public dataset included 3907 binary files. From this dataset, we only selected files which were less than 200kb (3575 files). The size was restricted in order to remove outliers files that would have skewed the size distribution of the data set.

Format	**ISA**	**Number of binaries**	**Maximum size**
ELF	ARM	3575	200kB

Table 1: Summary of raw binary dataset details.

Binary annotation To create our dataset, the first step was to disassemble the binary executables obtained made public by Clemens [1]. Since these files were structured based on one of the standard formats, namely ELF, we were able to use conventional binary analysis tools. We used IDA Pro [10] to disassemble these binaries. Using IDA Pro in terminal mode, we wrote a script to automatically load all the binary files into IDA one by one. For every file, we used the *idc* module to identify all the segments (See Fig. 2). These segments were then traversed and a label was generated for each byte (i.e., code or data). This was then written to a CSV file alongside their byte values and disassembly (See Tab. 2). The next step was to convert these reverse engineered CSV files into images. To label these images, we generated RGB images. In the RGB images, a pixel in position (x, y) represented whether the pixel in the same position in the visualized binary corresponded to a code or data byte. The code pixels were assigned the color green and data pixels were assigned the color red.

Visualization of binary files To visualize the binary files we assess the following aspects:



Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	T
.init	00010CD8	00010CE4	R	.	X	.	L	dword	01	public	CODE	32	00
.plt	00010CE4	00010F4C	R	.	X	.	L	dword	02	public	CODE	32	00
.text	00010F4C	00012088	R	.	X	.	L	dword	03	public	CODE	32	00
.fini	00012088	00012090	R	.	X	.	L	dword	04	public	CODE	32	00
.rodata	00012090	00012700	R	.	.	.	L	dword	05	public	CONST	32	00
.eh_frame	00012708	0001270C	R	.	.	.	L	dword	06	public	CONST	32	00
.init_array	00022EE4	00022EE8	R	W	.	.	L	dword	07	public	DATA	32	00
.fini_array	00022EE8	00022EEC	R	W	.	.	L	dword	08	public	DATA	32	00
.jcr	00022EEC	00022EF0	R	W	.	.	L	dword	09	public	DATA	32	00
.got	00023000	000230D4	R	W	.	.	L	dword	0A	public	DATA	32	00
.data	000230D4	000230F8	R	W	.	.	L	dword	0B	public	DATA	32	00
.bss	000230F8	00023194	R	W	.	.	L	qword	0C	public	BSS	32	00
extern	00023194	00023264	?	?	?	.	L	para	0D	public		32	00

Fig. 2: Segments of an ARM binary alongside their classes and address ranges.

Address	Class	Disassembly	Bytes
10326	Code	MOV R9, R0	129-70
59536	Code	ADD R5, SP, 0x8C+var_4	34-173
17494	Data	DBC 0xCC	204
38978	Code	STR.W LR, [SP, 0x78+var_34]	205-248-68-224
43118	Code	STR R5, [SP, 4]	1-149

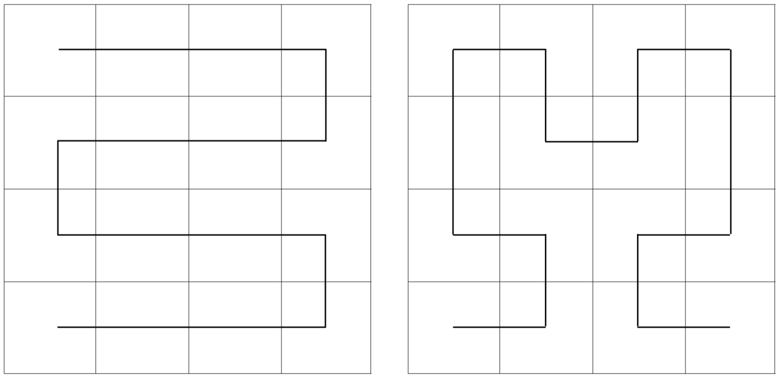
Table 2: Sample of CSV file content.

- **Visualization techniques:** To visualize the binary executable files we first convert the hex values and read it as a vector of 8-bit unsigned integers (this information is available in the CSV files created in the previous step). These integers can range from 0-255 and can be directly converted to pixels within an image. The resulting vectors can be converted into multi-dimensional arrays. In previous work concerning binary file visualizations both RGB and grayscale images were generated. RGB images encoded more information about the binary (e.g. entropy and size of sections). In our method we wanted to minimize the knowledge needed to train different models, we therefore just rely on the bytes values. Additionally, we just want to classify bytes into code or data, and therefore visualize our binaries into graysacle images. These grayscale images thus stored the values of all the bytes in a particular binary [14].
- **Traversal techniques:** As mentioned above, we converted binary into a vector of 8-bit unsigned integers. We then evaluated different methods used for mapping a vector into a 2-D plan. This was done to maintain proximity in a region of bytes when converted to an image, the intuition was that this would result in more accurate section class detection. We specifically looked at *Space filling curves(SFC)*. SFCs are commonly used to map a multi-dimensional space into a one-dimensional sequence, but the reverse process (i.e., a one dimentional array to multi-dimensional space) is also possible, for our purpose we used it for the latter. There are two major classifications to SFCs: *non-recursive* and *recursive*. Non-recursive SFCs include the Z-Scan

Curve and Snake Scan curve whereas recursive SFCs include the Hilbert Curve, Peano Curve and Gray Code Curve [28]. We chose Snake scan curves and Hilbert curves since they are the most effective in maintaining spatial proximity of bytes within the image:

1. Snake Scan: This traversal technique is a fairly simple one, it maps the binary row-by-row, but keeps the continuity intact (as opposed to the Z-Scan curve) by picking off at the ends of each row, hence creating alternating end points on each row (See Fig.3a).
2. Hilbert Curves: The Hilbert curve is a space filling curve that visits every point in a square grid with a size of any power of 2. The basic elements of the Hilbert curves are square structures with one open side and vectors that joins these square structures, these structures are what dictate the traversal order (See Fig.3b).

There are certain underlying properties of SFCs, such as coherence in continuity, clustering and preservation of direction. Although the snake-scan preserves direction relatively well, the Hilbert curves outperform in clustering and coherence in continuity. We show our experimental results later in the paper (See Section 4).



(a) Snake curve traversal path.

(b) Hilbert curve traversal path.

- **Dimension selection:** Since we convert the binary into a vector we also analyzed the optimal image dimensions. Since we are using SFC to convert the vector into a 2-D array each SFC method had to be assessed independently:
  1. Snake Scan: our initial intuition was to fix the width of images to 4 pixels/bytes since our dataset was comprised of a mixture of 32 bit ARM and ARM/Thumb binary files. Thumb mode instructions (which are 2 bytes instead of 4) would have also been accommodated for in the 4 pixel width scheme by duplicating the 2 pixels. However due to binary sizes, fixing the width of the image to 4 pixels/bytes resulted in images with extended lengths. Although this method would have ensured all rows of pixels/instructions are in close proximity to their preceding and

succeeding instructions, it does not maintain regional proximity, this is crucial for section class detection. Thus, this approach was discarded and we decided to use square images. The dimensions (width/height) were calculated as follows (See Fig.6a and Fig.6b):

$$dimension = \lceil \sqrt{total_{bytes}} \rceil \tag{1}$$

If the total number of bytes was not a perfect square, the image would have some pixels unfilled in the last rows. These unfilled pixels were filled with padding of black color in both the grayscale image and the RGB annotation images.

2. Hilbert curves: For Hilbert curves, we had to conform to square images since they can only be generated for square images with dimensions in the power of 2. Thus we first generated square images with Snake scan and then upsampled them to a width and height which was the nearest power of 2. The resulting image was then traversed pixel by pixel, according to the hilbert curves path, to generate the image (See Fig.7a and Fig.7b).

- **Size selection:** For upsampling, we analyzed the performance of several interpolation filters offered by the python library Pillow. Both Bilinear and Bicubic filters rely on interpolation to determine pixel values in the upsampled image. Upsampling using these aforementioned filters will result in creating new pixels/bytes that didn't exist in the original binary. This is problematic because not only does it create data but it also causes uncertainty in creating groundtruth class labels. Additionally, when this is applied directly on label images, this results in pixels which are not purely red/green since the values are calculated using weighted averages. Nearest neighbor interpolation, on the other hand, relies on the ratio of (x,y) coordinate in the original image to determine the position in the upsampled image. This method does not generate any new pixel values, rather it copies the values of the nearest neighbour. This is suitable for data that is not smooth (due to fine grained labeling). It also results in pixels which are purely green/red.

3.2 Semantic segmentation models

After generating the dataset of grayscale and their corresponding label images, the next step was to train a semantic segmentation model to learn to differentiate between code and data bytes and sections. To perform this binary analysis through images, we decided to use a semantic segmentation model, Deeplab [6].

Deeplab DeepLab is a state-of-art deep learning model for semantic image segmentation, where the goal is to assign semantic labels (e.g., person, background, tumor) to every pixel in the input image. While its primarily used to perform classification on natural images, we employ transfer learning to train the last

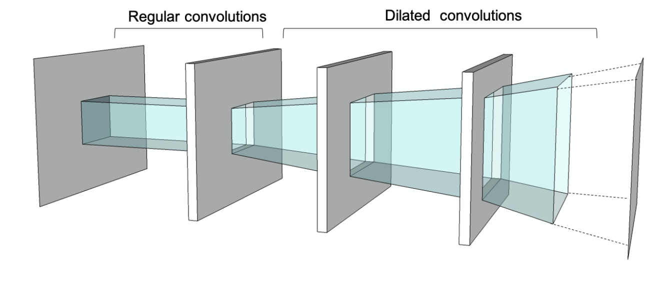


Fig. 4: Overview of CNNs with dilated convolutions used in DeepLab.v1.

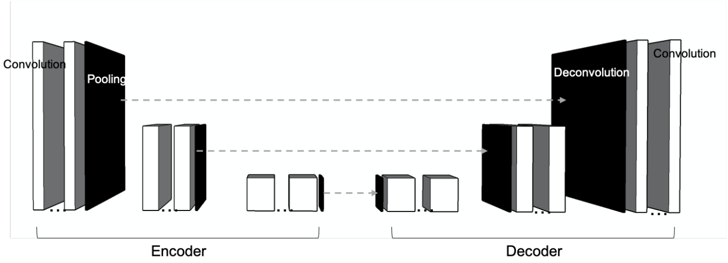


Fig. 5: Overview of encoder-decoder structure used in DeepLab.v3+.

few layers on our own binary files dataset. Deeplab.v1 is based on CNNs with dilated convolutions (See Fig. 4), the next version (v2) of DeepLab employed additional 'dilated (atrous) spatial pyramid pooling' (ASPP) layer and optimisation of ASPP layer hyperparameters. The final version (v3), the one used in this paper, leverages an encoder-decoder structure (See Fig. 5) with dilated convolutions [25].

	Learning Rate				**Iterations**			**Training batch**		
Hyperparameter value	0.0001	0.001	0.01	0.1	500	1000	2000	4	8	16
Accuracy	82.34	85.90%	86.53%	54.08%	88.64%	89.11%	89.68%	84.30%	85.25%	85.80%

Table 3: Hyperparameter evaluation for learning rate, number of iterations, and training batch size.

Tuning Hyperparameters We studied the hyperparameters of our baseline model in Deeplab and realized there was significant room for improvement. We tweaked variables one by one keeping everything else constant. This approach resulted in an increase in accuracy as we increased the number of iterations, learning rate, training batch size, and the dataset size. We tested out the accuracy against several pre-trained models on Deeplab. The model *deeplabv3xceptionade20ktrain* gave us the highest accuracy. We experimented with the model variants too, that are used to calculate the loss at every step; 'xception 65' generated the best results. Furthermore, models trained on Hilbert curve generated images showed better result compared to Snake scan images. This is due to the preserved locality when mapping from 1D to 2D space. Tables 3, 4 summarize the hyperparameters tweaked and the resulting improvement from each one.

	Output Stride		**Model variant**		**Pre-trained model**	
Hyperparameter value	16	32	xception_71	xception_65	xception_ade20k_train	pascal_train_aug
Accuracy	85.80%	84.80%	57.20%	85.80%	85.80%	83.70%

Table 4: Hyperparameter evaluation for output stride size, model variants, and pre-trained models used.

4 Experimental Evaluation

In this section, we evaluate our proposed methodology with respect to its separation accuracy. In particular, we evaluate the accuracy of our model in accurately classifying data and code sections and bytes for standard binary formats, namely, ELF 32-bit ARM binaries. We further use a real-world PLC dataset to study how well our model can adapt to non-standard binary formats.

4.1 Model performance on standard binary formats

Using the optimal hyperparameters found from the previous section, we evaluate the performance of our trained models on a subset of unseen data from the same dataset used for training. This is used to assess the proposed method's success on standard binary formats. For every image, we calculated the accuracy of its predicted image as follows:

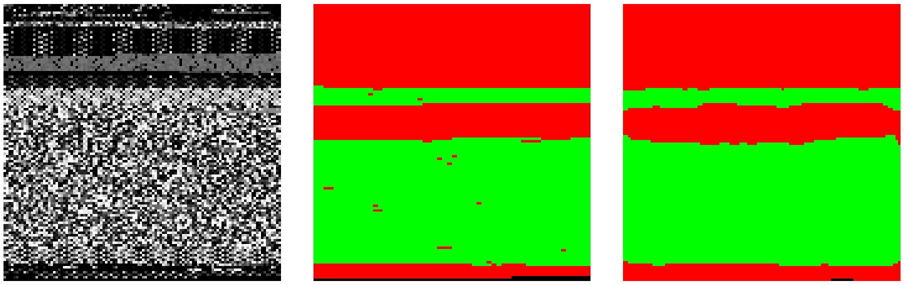
$$accuracy = (truepositives/total_{bytes}) * 100 \tag{2}$$

Here, the *truepositives* variable represents how many pixels were correctly identified (as code or data) while *totalbytes* indicates the total number of pixels in the image. The accuracies listed in the following section are derived by taking the average of accuracies of all images calculated using the aforementioned formula. Table 5 shows a summary of the results from our experiments. We can see that the model trained using Hilbert curve based images (max. accuracy: 91.91%) outperforms the model trained on Snake curve based images (max. accuracy:

89.22%) with a margin of nearly 3%. Additionally, Fig. 6 and Fig. 7 show the visual results from the experiments. It can be clearly seen from the original and experimental label image that the method performs very well for code section detection, while it under-performs for embedded bytes. (See Fig. 6b and Fig. 6c, Fig. 7b and Fig. 7c).

Traversal method	**Iter.**	**Learning rate**	**Dataset size (train)**	**Dataset size (test)**	**Image size**	**Accuracy**
Snake curve	**500**	**0.01**	**1999**	**304**	**256x256**	**89.22%**
	1000					86.53%
Hilbert curve	500					90.49%
	1000					**91.91%**

Table 5: Summary of results for standard binary formats using both Snake curve and Hilbert curve generated images.

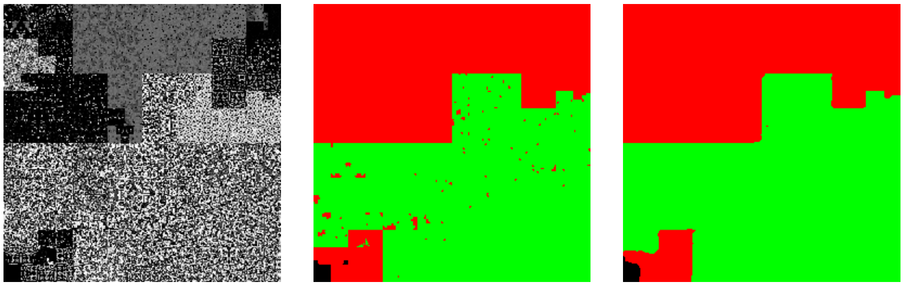


(a) Binary image. (b) Original label image. (c) Exp. label image.

Fig. 6: Overview of Snake curve generated binary images and results.

4.2 Model performance on non-standard binary formats

Using the best performing models for both Hilbert curve and Snake curve images, we evaluate the performance of our trained models on a dataset of real-world PLC binaries. This is used to assess the proposed method's success on non-standard binary formats. We obtained an open-source real-world PLC binary dataset [19]. After removing outlier files, the dataset contained 54 PLC binaries. These binaries are 32-bit ARM binaries, but do not conform to any standard binary format. We used these binaries as a proof of concept. Our experiments were limited to these binaries due to the challenging task of establishing ground-truth for byte annotations in non-standard binary formats. We used the same



(a) Binary image (b) Original label image. (c) Exp. label image.

Fig. 7: Overview of Hilbert curve generated binary images and results.

methods used in Sub-section 3.1 to visualize the binaries and to generate the ground-truth bytes/section label images. The only difference is that we used ICSREF [12] to find the true class labels for bytes and sections. Fig. 8 shows an overview of our testing setup.

We use the same method (See Eqn. 1) to calculate the accuracy of the model on PLC binaries. Our experiments show that there is a significant drop in accuracy when we test on non-standard binaries. Hilbert curve based models drop from 91.91% for standard binary formats to 81.75% for non-standard formats, while Snake curve based models drop 89.22% to 68.19%. Fig. 9 and Fig. 10 shed some light on the reason for the accuracy drop. It is evident that the PLC binary format contains a lot of inline data within code sections, the description of the binary format in ICSREF [12] mentions that the functions and function blocks sections contain data section at the end of each these sections. Since our method performed better with section identification, it was especially disadvantaged with this binary format. Still, we are able to clearly find the large sections and gain general insights about the new binary formats with minimal effort.

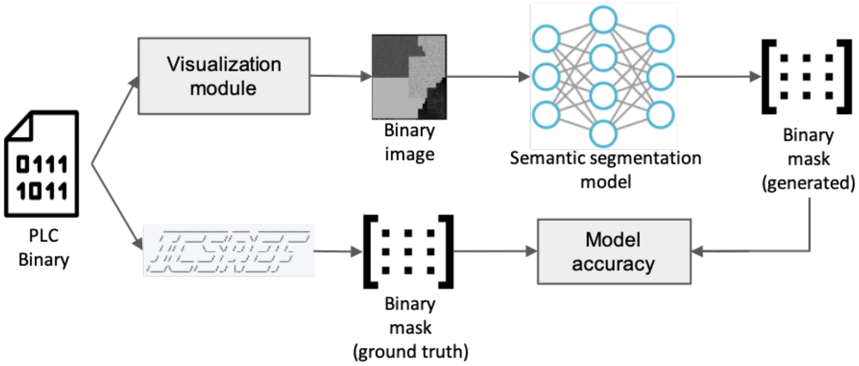
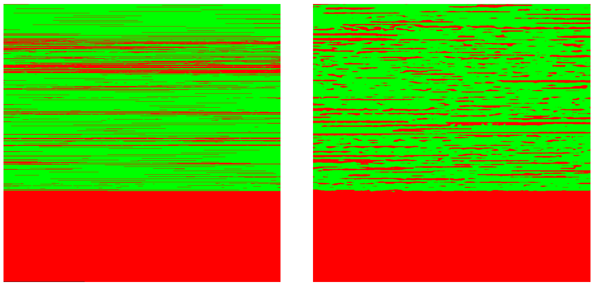
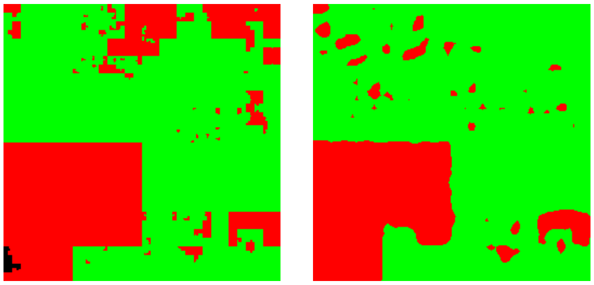


Fig. 8: Overview of the semantic segmentation testing Setup.



(a) Original label image. (b) Exp. label image.

Fig. 9: Overview of Snake curve generated binary images and results for PLC binaries.



(a) Original label image. (b) Exp. label image.

Fig. 10: Overview of Hilbert curve generated binary images and results for PLC binaries.

5 Related Work

Code and data separation in executable files is a well-known problem in static binary analysis. Conventional disassemblers have to perform this task as well. Many previous research efforts focused on detecting inline code within data sections or inline data within code sections for standard binary formats. Researchers in [27] focus on classification of bytes using a language model to identify the transitions from different data types (code to data, data to code, code to code) in x86 binaries. They leverage reference arrays and utility functions used to train the model. This work outperformed state-of-the-art solution IDA Pro. More recently researchers have started to utilise the advances in machine learning to approach this problem. The work in [11] uses supervised learning over a graph to solve the code discovery problem in x86 binaries, the authors of this paper leverage structural SVMs to classify bytes as code or data. Most of the work focuses on x86 binaries, this is due to their variable length instructions. [5] discusses the code discovery as a sub-problem of binary translation. The authors target ARM binaries and discuss the challenges introduced from ARM/Thumb binaries. The most recent paper in this field is [16], the authors propose ELISA a Code and Data section and byte class identification method. They analyze the binary as a sequence and use a Conditional Random Field (CRF) based model. They fortify their solution with ISA specific heuristics and an ISA detection method. Our work fills the gap concerning non-standard binary formats and focuses on building adaptable models that can extract information about these formats with no prior knowledge about the binary format.

6 Conclusion

The work in this paper proposed a new method for automating one of the stages of binary analysis, code and data separation. This automation is especially important and useful in the case of propriety/unavailable software source code. Although there exist classical binary analysis tools that are designed to analyze source code in an automated manner, most fail to analyze atypical binary formats. This is due to the fact that boundaries between data and code section are clear and conventional in typical binary codes. The correct and clear marking prevents any errors due to misinterpreting data as executable instructions and vice-versa. Due to the rapid changes and advancements in technology (i.e. Industry 4.0, IoT) more unknown formats of binary files are being produced. In this work we utilize image-based machine learning algorithms to automate our process. We convert the binary files to image files where the algorithm performs semantic segmentation to identify the data and code sections of the binary code. Our results show that the model trained using Hilbert curve-based images has a achieved a maximum accuracy of 91.91% while the model trained on Snake curve-based images achieved a maximum accuracy of 89.22%. The results also show that our suggested method performs very well for code section detection, while it under-performs for embedded bytes.

References

1. Automatic classification of object code using machine learning. Digital Investigation **14**, S156 – S162 (2015), the Proceedings of the Fifteenth Annual DFRWS Conference
2. Abbasi, A., Holz, T., Zambon, E., Etalle, S.: Ecfi: Asynchronous control flow integrity for programmable logic controllers. Proceedings of the 33rd Annual Computer Security Applications Conference (2017)
3. Abuhamad, M., Abuhmed, T., Mohaisen, A., Nyang, D.: Large-scale and language-oblivious code authorship identification. Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (2018)
4. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: Bap: A binary analysis platform. In: CAV (2011)
5. Chen, J.Y., Shen, B.Y., Ou, Q., Yang, W., Hsu, W.: Effective code discovery for arm/thumb mixed isa binaries in a static binary translator. 2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES) pp. 1–10 (2013)
6. Chen, L.C., Papandreou, G., Kokkinos, I., Murphy, K., Yuille, A.: Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. IEEE Transactions on Pattern Analysis and Machine Intelligence (2018)
7. Corina, J., Machiry, A., Salls, C., Shoshitaishvili, Y., Hao, S., Krügel, C., Vigna, G.: Difuze: Interface aware fuzzing for kernel drivers. Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (2017)
8. Cova, M., Felmetsger, V., Banks, G., Vigna, G.: Static detection of vulnerabilities in x86 executables. 2006 22nd Annual Computer Security Applications Conference (ACSAC'06) pp. 269–278 (2006)
9. Haller, I., Slowinska, A., Neugschwandtner, M., Bos, H.: Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In: USENIX Security Symposium (2013)
10. Kaner, Y., Sternberg, B.: Reverse-engineering database - an ida-pro plug-in (2012)
11. Karampatziakis, N.: Static analysis of binary executables using structural svms. In: NIPS (2010)
12. Keliris, A., Maniatakos, M.: ICSREF: A framework for automated reverse engineering of industrial control systems binaries. In: NDSS. The Internet Society (2019)
13. Khormali, A., Abusnaina, A.A., Chen, S., Nyang, D., Mohaisen, A.: Copycat: Practical adversarial attacks on visualization-based malware detection. ArXiv (2019)
14. Lee, D., Song, I.S., Kim, K.J., hyeon Jeong, J.: A study on malicious codes pattern analysis using visualization. 2011 International Conference on Information Science and Applications pp. 1–5 (2011)
15. Long, J., Shelhamer, E., Darrell, T.: Fully convolutional networks for semantic segmentation. 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) pp. 3431–3440 (2015)
16. Nicolao, P., Pogliani, M., Polino, M., Carminati, M., Quarta, D., Zanero, S.: Elisa: Eliciting isa of raw binaries for fine-grained code and data separation. In: DIMVA (2018)
17. Ronneberger, O., Fischer, P., Brox, T.: U-net: Convolutional networks for biomedical image segmentation. ArXiv (2015)
18. Rosenblum, N.E., Zhu, X., Miller, B., Hunt, K.: Learning to analyze binary computer code. In: AAAI (2008)

19. Sarkar, E., Benkraouda, H., Maniatakos, M.: I came, i saw, i hacked: Automated generation of process-independent attacks for industrial control systems. Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (2020)
20. Shoshitaishvili, Y., Wang, R., Hauser, C., Krügel, C., Vigna, G.: Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In: NDSS (2015)
21. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Krügel, C., Vigna, G.: Sok: (state of) the art of war: Offensive techniques in binary analysis. 2016 IEEE Symposium on Security and Privacy (SP) pp. 138–157 (2016)
22. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: Bitblaze: A new approach to computer security via binary analysis. In: ICISS (2008)
23. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Krügel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: NDSS (2016)
24. Tiegelkamp, M., John, K.H.: IEC 61131-3: Programming industrial automation systems. Springer (1995)
25. Ulku, I., Akagunduz, E.: A survey on deep learning-based architectures for semantic segmentation on 2d images. arXiv: Computer Vision and Pattern Recognition (2019)
26. Wartell, R., Zhou, Y., Hamlen, K.W., Kantarcioglu, M., Thuraisingham, B.: Differentiating code from data in x86 binaries. In: ECML/PKDD (2011)
27. Wartell, R., Zhou, Y., Hamlen, K.W., Kantarcioglu, M., Thuraisingham, B.: Differentiating code from data in x86 binaries. In: ECML/PKDD (2011)
28. Weissenbock, J., Fröhler, B., Gröller, E., Kastner, J., Heinzl, C.: Dynamic volume lines: Visual comparison of 3d volumes through space-filling curves. IEEE Transactions on Visualization and Computer Graphics **25**, 1040–1049 (2019)