

# Padding Matters – Exploring Function Detection in PE Files

Raphael Springer  
Westphalian University of Applied  
Sciences  
Institute for Internet Security  
Gelsenkirchen, Germany  
springer@internet-sicherheit.de

Alexander Schmitz  
Westphalian University of Applied  
Sciences  
Institute for Internet Security  
Gelsenkirchen, Germany  
schmitz@internet-sicherheit.de

Artur Leinweber  
Westphalian University of Applied  
Sciences  
Institute for Internet Security  
Gelsenkirchen, Germany  
leinweber@internet-sicherheit.de

Tobias Urban  
Westphalian University of Applied  
Sciences  
Institute for Internet Security  
Gelsenkirchen, Germany  
urban@internet-sicherheit.de

Christian Dietrich  
Westphalian University of Applied  
Sciences  
Institute for Internet Security  
Gelsenkirchen, Germany  
dietrich@internet-sicherheit.de

## ABSTRACT

Function detection is a well-known problem in binary analysis. While previous research has primarily focused on Linux/ELF, Windows/PE binaries have been overlooked or only partially considered. This paper introduces FuncPEval, a new dataset for Windows x86 and x64 PE files, featuring Chromium and the Conti ransomware, along with ground truth data for 1,092,820 function starts. Utilizing FuncPEval, we evaluate five heuristics-based (Ghidra, IDA, Nucleus, rev.ng, SMDA) and three machine-learning-based (DeepDi, RNN, XDA) function start detection tools. Among the tested tools, IDA achieves the highest F<sub>1</sub>-score (98.44%) for Chromium x64, while DeepDi closely follows (97%) but stands out as the fastest by a significant margin.

Working towards explainability, we examine the impact of padding between functions on the detection results. Our analysis shows that all tested tools, except rev.ng, are susceptible to randomized padding. The randomized padding significantly diminishes the effectiveness for the RNN, XDA, and Nucleus. Among the learning-based tools, DeepDi exhibits the least sensitivity and demonstrates overall the fastest performance, while Nucleus is the most adversely affected among non-learning-based tools.

In addition, we improve the recurrent neural network (RNN) proposed by Shin et al. and enhance the XDA tool, increasing the F<sub>1</sub>-score by approximately 10%.

## 1 INTRODUCTION

Binary code analysis is a building block of several applications addressing threats for Internet users, such as malware analysis or vulnerability research. One of the essential first steps in analyzing (compiled) applications is detecting functions in the code (e.g., as input to decompilation or for function-level similarity methods). For example, Haq and Caballero [11] find that 30 out of 61 binary similarity methods operate on a function-level granularity and thus rely on function detection as an initial analysis step. In this scenario, missing a function or incorrectly detecting a false function start might disrupt code similarity pipelines, potentially preventing them from correctly identifying a malware sample as part of a specific malware family. Further, without proper function detection, it is hard to assess the important parts for further analysis (e.g., to

identify the malicious capabilities a malware sample might exhibit). Hence, reliably extracting functions from compiled binary software is critical for analyzing binary code.

With millions of hash-unique malware samples emerging every year [14], automation is key to scalable analysis tooling in various use cases, e.g., binary similarity and clustering, malware lineage, actor and tool tracking for threat intelligence, or prioritization for dynamic analysis such as sandboxing [22]. Thus, reliable, fast, and automated function start detection is key.

Currently, existing tools often use heuristic or pattern-based methods to identify function starts [1, 20, 23]. These heuristics are compiled by experts based on their experiences when analyzing binaries. Like all heuristics, these approaches might be incomplete and must be updated regularly. Thus, recent work suggested machine learning-based approaches [7, 19, 25, 28] for function detection to increase performance and accuracy and reduce the need for experts to identify new patterns. However, Koo et al. [17] outline challenges and shortcomings when detecting functions in compiled binary code and revisit previous datasets, metrics, and evaluations. They show that previous work has suffered from effects such as overfitting (e.g., due to missing normalization), in the appropriate definition of true negatives, and imbalance due to significant redundancy in the datasets (shared static library), skewing the evaluation [17]. Thus, it is unclear whether machine learning-based approaches meet the expectations and can effectively reduce the need for human experts to develop heuristics. Additionally, related work often primarily covers functions in Linux/ELF samples and only analyzes fewer Windows/PE functions, if any. However, with a significant malware set targeting Windows operating systems [14], evaluating function detection on PE plays an important role. While the binary code parts of these two formats can use the same instruction set architecture (ISA) (e.g., x64), differences exist in the application binary interfaces (ABI) of Linux and Windows. Factors that may influence the function start detection efficacy include calling conventions and compiler-specific quirks (e.g., padding schemes, inline data, and metadata associated with Windows-specific tools like Microsoft Visual Studio). Thus, it is unclear if and how the proposed methods can be generalized from ELF samples to other formats.

In this work, we shed light on these challenges by comparing eight (five heuristics-based and three machine-learning-based) function start detection tools. We specifically focus on 32-bit and 64-bit Windows PE binaries of benign and malicious code examples. More specifically, based on previous works [19, 25], we train two machine-learning-based tools on a commonly used dataset [7] to find function starts. We then test the efficiency of these tools and six further tools [1, 5, 8, 9, 23, 28] on a newly built dataset consisting of samples and ground truth. We evaluate the tools on a Chromium sample for Windows and the *Conti* ransomware (x86 and x64). On a high level, our results indicate that all tested tools generalize well across file formats but that the effectiveness of machine-learning-based tools collapses when they face toolchain-specific quirks (i.e., different padding schemes).

In summary, we make the following contributions:

- We introduce FuncPEval, a new x86 and x64 Windows PE dataset that contains malicious and benign software samples spanning 1,092,820 functions. Using this dataset, we compare eight tools, proposed within the past decade and commonly used for function start detection in the field. We show that all used tools can generally identify function starts in regular PE files with high precision and recall.
- Based on previous work, we train two machine learning-based function start detection tools [19, 25] on a commonly used dataset. For further analysis, we optimize the provided methods, improving XDA’s  $F_1$ -score by approximately 10%.
- Finally, we demonstrate that modifying the padding between functions in a given sample impacts the function start detection. For some tools, the effectiveness of function start detection methods relies heavily on the unmodified padding between functions as emitted by standard compilers. When this padding is altered, the effectiveness for learning-based methods deteriorates, with  $F_1$ -scores down 30 to 70 percent points. Thus, our results indicate that the machine learning approaches might be susceptible to spurious correlation [6].

## 2 RELATED WORK

Function detection can be categorized into heuristics- and pattern-based detection (e.g., signatures on function prologues and epilogues), static analysis techniques (e.g., CFG extraction), and machine learning-based approaches described in more detail in the following subsections.

### 2.1 Static code analysis and pattern-based approaches

IDA Pro [23] uses proprietary patterns to identify function starts. Similarly, Ghidra [1] uses a combination of signatures<sup>1</sup> and static analysis techniques. The signatures typically cover machine code instruction sequences in the form of byte patterns that precede a function, such as padding or an epilogue, or that are typically observed at the start of a function, such as a prologue or allocation routines. In September 2022, Ghidra introduced the Random Forest

<sup>1</sup>Ghidra’s patterns are available from [https://github.com/NationalSecurityAgency/ghidra/blob/b9496de7f573e6a73888abfb51c243723785dbdb/Ghidra/Processors/x86/data/patterns/x86win\\_patterns.xml](https://github.com/NationalSecurityAgency/ghidra/blob/b9496de7f573e6a73888abfb51c243723785dbdb/Ghidra/Processors/x86/data/patterns/x86win_patterns.xml)

Function Finder Plugin, a machine learning-based approach that is trained on previously recognized functions of the currently analyzed binary and attempts to find similar function starts. This differs from the following machine learning approaches in that it is applied on a per-sample scope and does not attempt to globally model function starts. Andriess et al. [4] evaluate existing disassemblers and their function boundary recovery. They conclude that false negative rates of function starts typically reach 20% or more, indicating significant shortcomings when applied in practice. A follow-up work by Andriess et al. [5] proposes Nucleus, a compiler-agnostic function detection tool that constructs an inter-procedural control flow graph. Similarly, rev.ng by Di Federico et al. [9] proposes a set of analyses to extract function starts based on QEMU’s lifter and LLVM’s intermediate representation, thus operating without ISA-specific heuristics.

Alves-Foss and Song [2] propose detecting function boundaries using control flow analysis, jump and call targets, exception metadata, and detection of terminal and missing functions. Their approach is implemented in the Jima tool, which supports Linux ELF samples only and is currently only available in compiled form. FETCH by Pang et al. [18] leverages call frames, i.e., frame description entries in the exception handling information as mandated by the x64/amd64 System V Application Binary Interface. Such call frame information is typically added by the compiler at build time. While evaluating against x64 ELF samples only, the authors note that x64 PE and ARM will likely exhibit similar metadata. However, such metadata is not always present, especially in malware.

SMDA by Plohmann [8] combines recursive disassembly and heuristics for function entry point discovery and later performs a gap analysis to find missed functions.

In 2023, FunProbe by Kim et al. [15] proposes a probabilistic model using a Bayesian Network over causal relationships between heuristically identified function entry point candidates. First, an inter-procedural CFG is recovered, and up to 16 function identification hints are collected based on data-driven properties, e.g., FDE, and code-driven properties, e.g., call targets. Then, a Bayesian Network is built, followed by belief propagation. As a result, each byte yields an inferred probability of being a function start when exceeding a given threshold. FunProbe currently only supports ELF files and has been evaluated on a total of 19,872 ELF samples covering x86, x64, ARM, and MIPS architectures.

### 2.2 Machine learning based approaches

Approaches that leverage machine learning have initially been proposed by Rosenblum et al. [24]. They model function start detection as a classification problem using Conditional Random Fields.

In 2014, Bao et al. introduced a function boundary detection approach called ByteWeight [7]. It uses a weighted prefix tree to learn signatures that can be used to detect function starts. In the case of ByteWeight, each branch represents a sequence of bytes or instructions. The depth of the tree determines the length of the sequence. The weighted prefix tree adds a weight to each node in the tree, representing the probability that the branch starts a function. The authors build the weighted prefix tree by training a non-weighted prefix tree with all possible byte or instruction combinations using ground truth data. Therefore, ByteWeight suffers

**Table 1: Function start detection methods and evaluation papers. The amount of PE functions for Nucleus [5] is an estimate as the number of samples and the description in the paper indicate that the same dataset as in [4] was used. Compilers refers to the compilers that were used to generate the dataset, i.e., GNU Compiler Collection, Clang, Visual Studio, and Intel C/C++ Compiler.**

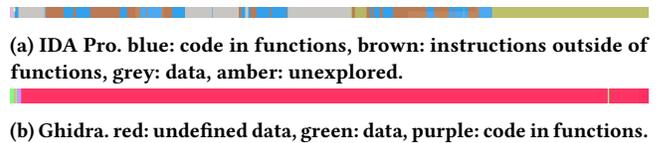
| Tool/Paper              | Compilers |       |    |     | Samples |     | Functions |              | ML |
|-------------------------|-----------|-------|----|-----|---------|-----|-----------|--------------|----|
|                         | GCC       | Clang | VS | ICC | ELF     | PE  | ELF       | PE           |    |
| FuncPEval (our dataset) |           | ✓     | ✓  |     | 0       | 4   | 0         | 1,092,820    | ✗  |
| FunProbe [15]           | ✓         | ✓     |    |     | 19,872  | 0   | 3,064,001 | 0            | ✗  |
| DeepDi [28]             | ✓         |       | ✓  |     | 1,440   | 688 | n/a       | n/a          | ✓  |
| Koo et al. [17]         | ✓         | ✓     |    |     | 152     | 0   | 769,069   | 0            | ✗  |
| FETCH [18]              | ✓         |       |    |     | 43      | 0   | 1,105,278 | 0            | ✗  |
| XDA [19]                | ✓         |       | ✓  | ✓   | 2,593   | 528 | n/a       | n/a          | ✓  |
| Jima [2]                | ✓         | ✓     |    | ✓   | 3,790   | 0   | 4,913,753 | 0            | ✗  |
| LEMNA [10]              | ✓         |       |    |     | 2,064   | 0   | n/a       | 0            | ✓  |
| Nucleus [5]             | ✓         | ✓     | ✓  |     | 324     | 152 | n/a       | est. 378,965 | ✗  |
| REV.NG [9]              | ✓         | ✓     |    |     | 1,890   | 0   | n/a       | 0            | ✗  |
| Andriess et al. [4]     | ✓         | ✓     | ✓  |     | 829     | 152 | 1,525,024 | 378,965      | ✗  |
| Shin et al. [25]        | ✓         |       | ✓  | ✓   | 2,064   | 136 | 598,359   | 187,836      | ✓  |
| BAP/ByteWeight [7]      | ✓         | ✓     |    |     | 2,064   | 136 | 598,359   | 187,836      | ✓  |
| Rosenblum et al. [24]   | ✓         |       | ✓  | ✓   | 728     | 443 | 283,626   | 100,427      | ✓  |

from the same problems as traditional signature-based approaches, e.g., depending on compiler versions. Nevertheless, ByteWeight can automatically generate signatures when ground truth data is available. For each architecture and compiler, ByteWeight has to generate new signatures and, therefore, also requires new ground truth data. While ByteWeight aims to detect function starts, Bao et al. present further analysis techniques that can be applied after the function start detection that lift the approach to detect all instructions belonging to the function.

In 2015, Shin et al. [25] introduced function boundary detection using a bidirectional recurrent neural network (RNN). The RNN uses a sequence of bytes as input and decides for each byte if it marks the start of a function. Similar to ByteWeight, the weights of the RNN are trained using ground truth data. The trained weights form the model used to detect function starts. The RNN can also detect the boundaries of a function by using two models. One model detects function starts, and the other model detects function ends. The authors do not provide an implementation of their approach. However, a reimplementation was provided by Guo et al. [10] as part of their work on the explainability of machine learning-based methods.

In 2021, Pei et al. [19] propose XDA, which relies on transfer learning of machine code disassembly and also recovers function boundaries. They are motivated by masked language modeling to infer dependencies between specific bytes in machine code. While the paper evaluates on x86 and x86-64 samples, targeting both ELF and PE, a fine-tuned model of XDA has only been published for x86-64. XDA models function boundary detection as a multi-class classification problem where a specific byte can either form the start of a function, the end of a function, or neither of both.

DeepDi, a system published by Yu et al. [28] in 2022, combines instruction-level sequences with a graph convolutional network to achieve disassembly. First, given a byte string as input, all possible instructions are decoded using a 15-byte sliding window over the



**Figure 1: Comparison of code, data, and unexplored areas in memory-mapped views of a Urausy malware sample.**

input stream, yielding the superset of instructions. Then, an instruction flow graph is constructed that captures the most likely true instructions and their relations. The system also contains heuristics and a classifier for function start detection based on the resulting disassembly. In contrast to previous work, DeepDi is the first learning-based approach that operates on the instruction level instead of the byte level.

### 2.3 Limited focus on PE files in existing work

Table 1 summarizes the related work, including details such as the number of samples and functions used for evaluation, where available. These figures are presented separately for ELF and PE binaries. The table reveals that only 6 out of 13 studies evaluated PE samples. Comparing the number of samples and functions between ELF and PE in these six cases highlights a significant underrepresentation of PE binaries in the existing literature. Given the importance of function detection in PE samples, particularly in the context of malware analysis [14], it becomes evident that a new evaluation focusing on PE binaries with a larger set of functions is necessary.

## 3 BACKGROUND ON FUNCTION DETECTION

Function detection in compiled code is not straightforward. Different approaches to function detection are likely to yield varying

sets of recognized functions stemming from the diverse methodologies employed. The following example highlights the significant differences in the detection of functions in a malware analysis context. Figure 1 shows two memory mapped representations of a Urausy malware sample<sup>2</sup>, produced by IDA Pro [23] and Ghidra [1], two popular reverse engineering tools. The colors represent the types of data or code when mapped in memory. The tools differ significantly regarding the detected functions. Figure 1a shows the analysis results of IDA Pro where blue-colored areas represent detected functions, brown-colored areas represent instructions that do not belong to functions, gray-colored areas represent data, and amber-colored areas represent unexplored areas that could not be further specified by IDA. In contrast, Figure 1b shows the analysis results of Ghidra where purple-colored areas represent detected functions, green-colored areas represent data, and red-colored areas represent undefined data. The figures show that the tools differ significantly in the detected functions in this sample. Manual analysis of this sample reveals that the instructions that IDA classified as not belonging to functions actually belong to functions. Similarly, Ghidra misclassified such code as undefined data. In practice, an analyst would need to inspect the code and manually define functions that were missed during function start detection, a labor-intensive task. The example shows that the function detection problem is far from being solved and that an evaluation of the existing tools is appropriate and needed.

Following most related work, we refer to function detection as finding bytes in compiled binary code that belong to the same function in the source code and for which no symbol information is given. There are other terms that refer to the same problem, e.g. function boundary detection, function boundary identification, function identification, and function recognition [5, 7, 17]. We will use the term *function detection* to avoid confusion with library function recognition [21, 26]. This different problem may occasionally also be referred to as function identification and describes recovering the semantic meaning of a function given its binary code, which is out of the scope of this paper.

Furthermore, we distinguish between i) function start detection, which only considers the *start* of functions, and ii) function boundary detection, which typically detects function *start* and *end* addresses, and iii) code ranges covering intervals of function code, most relevant for non-continuous functions.

Unless stated otherwise, we focus on function *start* detection for the remainder of this work because it applies to all related work and thus allows us to include the most tools in our evaluation. Furthermore, it can be modeled as a binary classification problem, enabling the use of well-known and considered evaluation metrics.

To evaluate function start classifiers, we use the *precision*, *recall*, and *F<sub>1</sub>-score* as metrics. To illustrate, Figure 2 depicts a schematic, fictional binary of size 24 bytes, alongside a hypothetical classification of function starts. Here, the ground truth marks the bytes at offsets 6 and 10 as valid function starts (e.g., obtained via debugging symbols). In contrast, the hypothetical classifier considers the bytes at offsets 4 and 10 to be function starts. True positives (TP) refer to bytes correctly identified as function starts, matching the

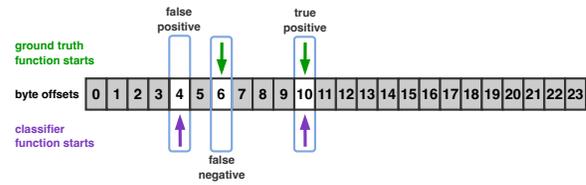


Figure 2: Schematic of function start detection

ground truth, e.g., the byte at offset 10 in Figure 2. False positives (FP) are bytes incorrectly classified as function starts, as they do not correspond to function starts in the ground truth (e.g., byte 4 in Figure 2). False negatives (FN) are bytes that are not classified as function starts but are function starts according to the ground truth (e.g., all grey-shaded bytes in Figure 2).

Note that in Figure 2, only two out of 24 bytes represent function starts. This highlights a common characteristic: the number of function starts rarely exceeds a small fraction of the total number of bytes in the file. Since a binary consists of much more than function starts, and each function start is represented by only a single byte, this imbalance is likely prevalent in most datasets. Such imbalance must be accounted for during training (e.g., by initializing the biases accordingly) and evaluation (e.g., by avoiding accuracy as a metric).

As previously described, most bytes in our context will not represent a function start. Given this imbalance, we avoid *accuracy* as a metric because it factors in true negatives in the computation. The example in Figure 2 would yield an accuracy of 92%, and an *F<sub>1</sub>-score* of 50%. Similarly, while practically useless, a naive classifier that predicts *every* byte as a non-function start would still result in a (comparatively) high number of true negatives and, consequently, high accuracy. Arp et al. [6] refer to such a pitfall as inappropriate performance measures.

## 4 EMPIRICAL VALIDATION

Our work aims to assess all function start detection tools introduced within the past decade that can operate on PE files. To achieve this, we describe the BAP/ByteWeight dataset as it is typically used to train models, and develop models for two learning-based approaches. We reproduce the work of Shin et al. [25], as their original implementation is not publicly available, and existing reimplementations by other researchers [10, 19] do not include trained models. Therefore, we provide a stable, well-documented implementation of Shin’s RNN-based classifier and publish the x86 and x64 Windows PE training data along with the trained models.

Additionally, we discovered a discrepancy in the implementation of XDA [19], leading to the reproduced results deviating from those reported in the original paper. To rectify this inconsistency, we introduce a novel encoding for the training data, leading to results that more closely align with those presented in the paper.

### 4.1 BAP/ByteWeight Dataset

To reproduce the original implementations of Shin’s RNN and XDA, we use the same dataset for training and evaluation, as used in the original studies. The dataset was initially compiled as part

<sup>2</sup>SHA256 hash value of 8f4296a0990ec245997bd2bb75edb512aae4e544b7d0c36e945bf19241fda426

of BAP/ByteWeight [7]. We only use a subset of the dataset, i.e., PE binaries targeting Microsoft Windows and their corresponding ground truth files containing the function starts as virtual addresses (VA). In the following, we refer to it as BAP dataset. This dataset spans a total of 136 hash-unique PE samples, built using Microsoft Visual Studio versions 2010 to 2013. It consists of 68 x86 and 68 x64 PE samples compiled from 17 programs (7z, vim, various PuTTY tools, hidapi, libsodium, sfxsetup, smtpsend), using four optimization levels (Od, O1, Ox, O2). In one case (filename `msvs_whatever_32_Od_SfxSetup`), the ground truth in the dataset did not match the provided sample as it contains a function VA where no section is mapped in memory. This could potentially be due to a misunderstanding or an error made by the original authors. To avoid label inaccuracy [6], we exclude this specific x86 sample from our dataset, resulting in a total of 135 samples (67 x86 and 68 x64).

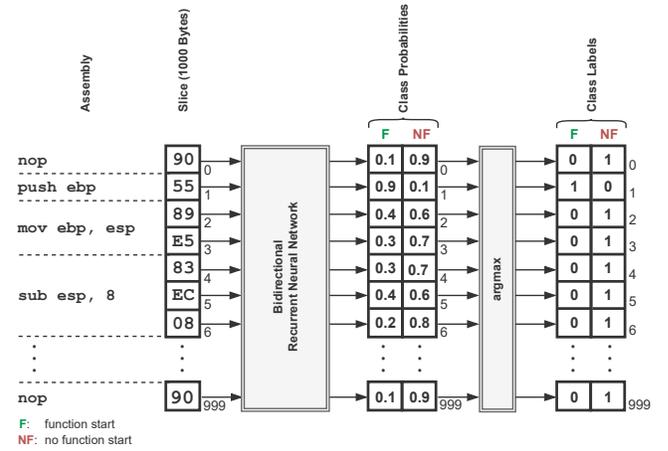
Koo et al. [17] highlight a pitfall of the BAP/ByteWeight dataset, particularly in the ELF subset, that we do not use. When normalizing instructions by blinding immediates and call and jump targets, only 17.6K (12.1%) out of the whole 146K functions of the ELF subset form unique normalized functions [17]. In other words, without normalization, overfitting becomes likely, especially when a significant number of normalized functions is present in both the training and the validation set. We decided against normalization in the training part of our pipeline to keep our results comparable with prior work. However, we consider uniqueness and normalization in our new FuncPEval dataset introduced in Section 5.

## 4.2 Implementation of the RNN-based Classifier

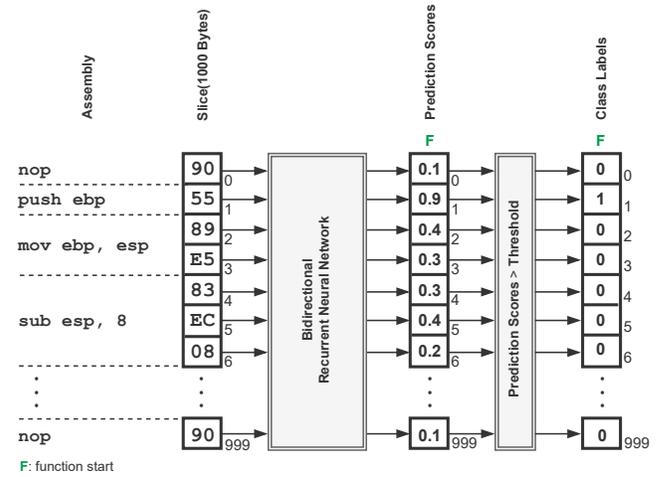
Currently, no public implementation of Shin’s RNN includes trained models for the classification of PE files. To incorporate the RNN into our evaluation in Section 5, we reproduce the original implementation and train models for x86 and x64 PE. Based on previous work by Shin et al. [25] and using the LEMNA reimplementation by Guo et al. [10] as basis, we implement the RNN-based classifier in Python using Keras 2.8.0 and TensorFlow 2.8.0 as backend. Considering the function start detection as a binary classification problem, we also implement a second variant with a slightly modified pipeline, namely one output neuron. In contrast, the LEMNA reimplementation uses two output neurons (Figure 3a). Our modification allows us to adapt the trigger threshold and thereby improve the classification results as shown in Table 2. The LEMNA-based RNN pipeline and our modified pipeline are shown in Figure 3.

Shin’s RNN architecture specifies that a sample must be divided into 1000-byte slices. If the last slice is shorter than 1000 bytes, the slice will be padded with zeros. Subsequently, the slice is fed into the RNN, where i) the LEMNA-based RNN outputs two class probabilities for each byte, and ii) our modified RNN yields a prediction score for each byte. Thus, we can predict for each input byte if it is the start of a function in the binary. Finally, a mathematical function assigns a class label to the byte. Guo et al. [10] use *argmax*, which determines the class label based on the maximum of the two class probabilities. Our variant uses a heuristic thresholding function:

$$C_{Label}(P_{Score}) = \begin{cases} 0 & \text{if } P_{Score} \leq t \\ 1 & \text{if } P_{Score} > t \end{cases} \quad (1)$$



(a) LEMNA-based RNN pipeline with two output neurons



(b) Our modified RNN pipeline with one output neuron

Figure 3: Comparison of the original and the modified RNN pipelines

If the prediction score  $P_{Score}$  exceeds the threshold  $t$ , the class label  $C_{Label}$  for the byte will be set to 1 (i.e., function start), otherwise 0 (i.e., not a function start). To determine the optimal threshold, the training dataset was evaluated with threshold values from 0 to 1 at intervals of 0.01. Subsequently, the threshold (0.38) with the best  $F_1$ -score was selected and used for the remaining experiments.

Note that when considering function starts on a per-byte level, every dataset containing real-world binaries is imbalanced as shown in Section 3, because bytes that do *not* represent a function start appear more often than those that *do* represent a function start. We randomly initialize the weights of the RNN and take the imbalance into account by changing the bias of the output layer. The initial bias  $b_0$  is computed as follows:

$$b_0 = \log_e \left( \frac{pos}{neg} \right) \quad (2)$$

where  $pos$  is the number of bytes representing the start of a function and  $neg$  is the number of bytes that are *not* the start of a function. Shin et al. [25] use the adaptive learning rate optimizer RMSprop [12] for training, while we use Adam [16], the same used in LEMNA.

**4.2.1 Training and Validation.** We train and validate separately for x86 and x64 PE samples. Given the BAP dataset of 67 x86 PE samples, we perform 10-fold cross-validation as follows: Split the dataset into ten (nearly) equally-sized disjoint subsets where each subset spans ca. 10% of the samples.

In each fold, executable sections from 9 of the 10 subsets (ca. 90% of the samples) are used for training, and executable sections from the remaining subset (ca. 10% of the samples) are used for validation. Shin et al. use a batch size of 32, while Guo et al. [10] used 100. We use a batch size of 1,000 for the given dataset to increase training speed. While Shin et al. trained for two hours, we followed LEMNA and trained the RNN for 150 epochs for each fold (taking approximately 55 minutes per fold). We run our experiments on a server with two 2.20 GHz Intel Xeon Silver 4114 CPUs à 20 threads and 128 GB of RAM.

**4.2.2 Evaluation.** Table 2 shows the precision, recall and  $F_1$ -score, on average. Although they do not exactly match the values from the previous work by Shin et al., we consider the results similar, indicating that our implementation provides a suitable RNN-based function start classifier. Note that our modified variant, referred to as one output neuron RNN, achieves higher  $F_1$ -score values for both x86 and x64, compared to the two output neurons RNN. As a result, we use the one output neuron RNN in subsequent experiments.

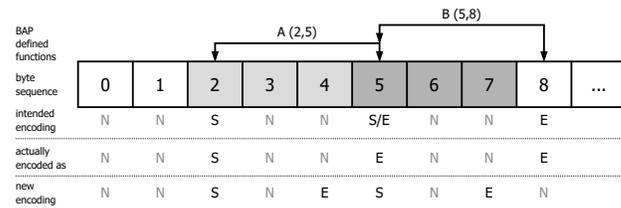
In the original paper, Shin et al. yield higher  $F_1$ -scores compared to our models. However, we lack a precise explanation, as their implementation and trained models are inaccessible. This may be an artifact of different folds or implementation details. For the experiments in Section 5, we train models for PE x86 and x64 using the whole BAP dataset. We publish our documented implementation, training data, and models.

### 4.3 Improving XDA

In preparation for the tool comparison in Section 5, we noticed discrepancies in the encoding of function starts and ends in the published XDA tooling. Consequently, we reproduce the results to verify their correctness. We utilized the model and code published by the authors for 64-bit PE files and applied it to the x64 part of the BAP dataset (BAP-64). In the original evaluation, 90% of the BAP dataset was used for evaluation because 10% was used to train XDA. Since we do not know which samples were used for training and which for evaluation, we use the entire dataset for evaluation. This should only positively impact the results for XDA, as 10% of the dataset was already seen during training. First, the function starts and ends are predicted using the provided model. Subsequently, we combine these starts and ends into function boundary pairs using the published algorithm<sup>3</sup>. Following this, we utilized the  $F_1$ -score calculation provided by the authors<sup>4</sup> to closely align with

<sup>3</sup>[https://github.com/CUMLSec/XDA/blob/main/scripts/play/eval\\_pair\\_bound.py#L6](https://github.com/CUMLSec/XDA/blob/main/scripts/play/eval_pair_bound.py#L6), Commit: c3ce2f

<sup>4</sup>[https://github.com/CUMLSec/XDA/blob/main/scripts/play/eval\\_pair\\_bound.py#L39](https://github.com/CUMLSec/XDA/blob/main/scripts/play/eval_pair_bound.py#L39), Commit: c3ce2f



**Figure 4: Labeling of Functions in XDA. For two adjacent functions, XDA would need to assign two labels for one byte. That is impossible, and a new encoding is required.**

the original evaluation. The computed  $F_1$ -score is shown in Table 3 in the column named *our experiment*, and the  $F_1$ -score from the original work in the column named *reported*.

We also include Nucleus, Ghidra, and IDA in the evaluation to provide a better comparison with the original evaluation [19]. We chose not to include the bi-RNN, as the XDA authors did not provide trained models for its implementation, making a fair comparison impossible.

The results in Table 3 indicate that the outcomes of our evaluation for IDA and Ghidra are similar to those in the original evaluation. The slight deviation could be attributed to the use of a slightly different dataset (100% of BAP-64 in our evaluation instead of 90%). However, there is a significant discrepancy between our evaluation and the original evaluation in the results for XDA (~17 percent points in  $F_1$ -score) and Nucleus (~10 percent points in  $F_1$ -score). To investigate the cause of this discrepancy, we conducted a detailed analysis of XDA’s detection mechanism.

**4.3.1 XDA Label Encoding.** Upon closer examination of the functions detected by XDA, we noticed that XDA fails to correctly identify functions when they are immediately adjacent to each other, i.e., when one function ends and another begins directly afterward without any bytes in between. This issue stems from the labeling of functions in the BAP dataset. In the BAP dataset, each function is labeled with a start and an end. The start is inclusive, marking the first byte of the function, while the end is exclusive, indicating the first byte that does not belong to the function. When one function directly follows another without intervening bytes, the end of the first function is the same as the start of the second function in the BAP notation. XDA assigns exactly one label per byte,  $S$  for a function start,  $E$  for a function end, and  $N$  for neither. In the scenario of two adjacent functions, XDA either correctly identifies the end of the first function *or* the start of the second function. Therefore, XDA cannot identify both functions in such cases. Figure 4 illustrates this issue. Function B (5,8) directly follows function A (2,5). XDA would need to assign both, the label  $S$  and the label  $E$ , to byte 5 to identify both functions correctly. If XDA classifies byte 5 as the end of a function, there are two function ends, i.e., byte 5 and byte 8, without a function start in between, and XDAs pairing code yields the function boundary pair (2,8), effectively representing a single function starting at byte 2 and ending at byte 8. This would result in one false positive and two false negatives in the evaluation. Note that we do not consider this a shortcoming of the XDA classifier but rather a labeling issue.

**Table 2: Results from 10-fold cross-validation of our RNN models compared to previous work by Shin et al., using the BAP PE dataset. Shin et al. report higher F<sub>1</sub>-scores, but their implementation and trained models are unavailable for inspection.**

| Method                 | PE x86    |        |                       | PE x64    |        |                       |
|------------------------|-----------|--------|-----------------------|-----------|--------|-----------------------|
|                        | Precision | Recall | F <sub>1</sub> -score | Precision | Recall | F <sub>1</sub> -score |
| Shin et al. [25]       | 99.01%    | 98.46% | 98.74%                | 99.52%    | 99.09% | 99.31%                |
| two output neurons RNN | 97.41%    | 92.42% | 94.83%                | 98.66%    | 96.43% | 97.53%                |
| one output neuron RNN  | 96.96%    | 95.65% | 96.29%                | 98.62%    | 98.29% | 98.45%                |

**Table 3: Improving XDA on the BAP-64 PE dataset. The column *our experiment* shows the results in our reproduction, the column *reported* shows the results as presented in [19], and *adapted GT* shows the results evaluated against the adapted ground truth, which was potentially used in [19]**

| Tool             | F <sub>1</sub> -score (PE x64) |               |            |
|------------------|--------------------------------|---------------|------------|
|                  | our experiment                 | reported [19] | adapted GT |
| IDA              | 91.13%                         | 90.5%         | 78.66%     |
| Ghidra           | 78.69%                         | 80.6%         | 71.22%     |
| Nucleus          | 79.80%                         | 70%           | 67.55%     |
| XDA reproduced   | 82.68%                         | 99.4%         | 97.81%     |
| XDA new encoding | 93.66%                         | -             | -          |

In the published artifacts, this issue also affects the training data for XDA. Each byte is assigned exactly one label, and the label *E* is assigned in case of a label conflict. Consequently, XDA never encounters a function start that immediately follows the end of another function during training. Additionally, the published code<sup>5</sup> suggests that the ground truth data for the evaluation was encoded similarly, causing the ground truth in XDA to deviate from the BAP ground truth. We attempted to reconstruct the ground truth as used in the original work by i) dividing the set of functions in the BAP ground truth into pairs of function starts and ends, ii) extracting all function starts in the set of function starts that also appeared in the set of function ends, and iii) forming new function boundary pairs by using the published algorithm<sup>6</sup>.

With the adapted ground truth, the evaluation results for XDA and Nucleus are much closer to those in the original evaluation, as shown in Table 3 in the rightmost column named *adapted GT*. The minor discrepancies could be attributed to the slightly different datasets used (100% of BAP-64 in our evaluation instead of 90%). However, the results for IDA and Ghidra deviate significantly from the original evaluation compared to the BAP ground truth. Although speculative, one possible explanation is that different tools might have been evaluated differently, using the adapted ground truth for XDA and Nucleus while using the original BAP ground truth for IDA and Ghidra.

**4.3.2 Retraining XDA.** We aim to address the labeling inaccuracy of the training data encoding by slightly modifying the training

<sup>5</sup>[https://github.com/CUMLSec/XDA/blob/5315918317eda39bf5de8ca56935baabfc30aa7e/scripts/play/eval\\_pair\\_bound.py#L144](https://github.com/CUMLSec/XDA/blob/5315918317eda39bf5de8ca56935baabfc30aa7e/scripts/play/eval_pair_bound.py#L144), Commit: c3cce2f

<sup>6</sup>[https://github.com/CUMLSec/XDA/blob/main/scripts/play/eval\\_pair\\_bound.py#L6](https://github.com/CUMLSec/XDA/blob/main/scripts/play/eval_pair_bound.py#L6), Commit: c3cce2f

data and retraining XDA. In the new encoding, we treat the end of a function as inclusive, just like the start, marking it as the last byte that still belongs to the function (*new encoding* in Figure 4). During evaluation, we account for this new encoding method by increasing the value of each function end by one. This adjustment is also applied when evaluating Ghidra and Nucleus, as both also consider the end of the function to be inclusive. With the help of the new encoding, only labels for one-byte-sized functions would result in a label conflict. In these cases, it is impossible to assign a single correct label, as the first and only byte of the function represents both the start and the end. To address this, a new label would need to be introduced to mark a function’s start *and* end. Since single-byte-sized functions are extremely uncommon, only occurring in about 0.1% of the functions in the BAP-64 dataset, we decided not to implement this change.

We use a randomly selected 10% of the samples from the BAP-64 dataset to retrain (fine-tuning part only) XDA and evaluate the entire BAP-64 dataset. This should not negatively impact the results of XDA, as 10% of the evaluation dataset was already seen during training.

This newly trained version of XDA achieves significantly better results than the original version, as shown in the last row of Table 3, with an increase from 82.68% to 93.66% in the F<sub>1</sub>-score. The F<sub>1</sub>-score of the newly trained version is still more than 5 percent points lower than the value reported in the original paper [19]. However, in our evaluation, the retrained version of XDA again emerges as the tool with the best results compared to IDA, Ghidra, and Nucleus. This supports our assumption that the XDA approach fundamentally works well, although it does not quite meet the claims of the original paper. To investigate this further, the evaluation could be repeated on other datasets, i.e., SPEC2006, SPEC2017, and the x86 versions. However, the pre-trained models used in the original evaluation would need to be made available for a fair comparison. Our experience underlines the importance of reproducible artifact and dataset publication in our research community to better explain such differences.

**Lessons learned.** This section shows how Shin’s RNN can benefit from using a one-output-neuron pipeline yielding F<sub>1</sub>-scores of 96.29% for PE x86, and 98.45% for PE x64. Similarly, XDA significantly benefits from a different labeling scheme, improving its F<sub>1</sub>-score by nearly 11 percent points for PE x64. Adjusting the labeling to account for adjacent functions underscores the critical role of thoroughly understanding and modeling the problem domain.

## 5 COMPARING FUNCTION START DETECTION TOOLS

As demonstrated in Section 2, prior research primarily focused on evaluating function start detection tools using ELF binaries. In contrast, our objective is to evaluate function start detection tools developed over the past decade exclusively on PE binaries. We introduce and release a new Windows PE dataset called *FuncPEval* to facilitate this. This new PE evaluation dataset spans 549k functions (233k normalized functions) for x86 and 543k functions (316k normalized functions) for x64, more than twice compared to previously available PE datasets. As a result, our dataset allows comparing tool performance using  $F_1$ -score and execution speed. Additionally, we investigate the impact of padding bytes between functions on function start detection.

### 5.1 FuncPEval

To address the limitation of prior research focusing predominantly on ELF binaries, we introduce a new dataset, *FuncPEval*, which exclusively comprises PE binaries. The dataset contains PE binaries targeting Microsoft Windows, stripped off their debugging information. As benign software, the core library `chrome.dll` of Chromium version 109 is chosen, both as x86 and x64 PE. These samples were compiled and linked by Google using LLVM clang 14.10.25019, using various per-module optimization levels, including Ox, Os, Oy, O1, Ot, and O2. For x64, we use Chromium snapshot 1069922<sup>7</sup>, and for x86, we use Chromium snapshot 1069956<sup>8</sup>, both released 10 Nov 2022. We choose to include Chromium for two main reasons: First, given its wide adoption, relevance in practice, and diverse code base, Chromium is a suitable target for binary code analysis. Second, Chromium for Windows has not been used in previous evaluations, rendering it unlikely that existing tools have been particularly optimized for our dataset.

We extract ground truth on the function start addresses from the associated PDB files using Microsoft’s DIA API and consider functions that are designated as symbol type *Function* and are listed under the respective compiled modules (`*.obj`). For each module, relative virtual function addresses (RVA) and other symbols (*FuncDebugStart*, *FuncDebugEnd*, etc.) are specified, which can be found in the PE file after linking the respective modules. We ignore all other symbol types, e.g. *Thunk*, because they do not provide any relevant information in the context of the given problem.

In addition to Chromium, we evaluate against the code of the Conti ransomware, a prevalent malicious software made publicly available by a leak in early 2022 [27]. We compiled and linked *Conti* version 3 using Visual Studio 2022 (version 14.34.31933) for both x86 and x64 and generated PDB files to obtain the ground truth. In the following, we consider the crypter (`cryptor.exe`), which is the component that encrypts files on the victim machine. To the best of our knowledge, no prior work has evaluated function start detection using malware in combination with reliable ground truth. While it would be beneficial to include a broader range of

samples and malware families, obtaining reliable ground truth is challenging, as it necessitates access to either a compilable version of the source code or corresponding debugging symbols.

To highlight the diversity in our evaluation dataset, we analyzed the binary code as shown in Table 4. For example, Chromium x64 exhibits 542,902 distinct RVAs in its `chrome.dll` that denote the start of a function. However, these represent 536,182 byte-unique functions, i.e. some functions are byte-equal multiples. ByteWeight [7] proposed normalizing instructions by removing immediates and call/jump targets. When normalized, the Chromium x64 sample contains 315,745 distinct normalized functions (roughly 58% of all functions). This shows that Chromium exhibits enough diversity for our evaluation. For reference, the overlap between Chromium x64 and BAP-64 is minimal, consisting of only 95 byte-unique functions (out of 601,820), corresponding to 176 normalized functions.

In addition, we analyzed the function prologues for the x64 samples. Assuming that function start detection tools may consider the bytes at the beginning of a function, which often represent the function prologue, particularly important, their diversity in the dataset becomes particularly noteworthy. Hence, the second-rightmost columns in Table 4 describe the diversity of prologues. Information about the prologue is derived from the `.pdata` section in x64 PE files, and only available for functions that use exception handling<sup>9</sup>. For Chromium x64, a non-zero-sized prologue was present in 470,317 functions, representing 7,488 unique prologue byte sequences. When normalizing the prologue instructions, 1,982 distinct sequences remain. While this number is significantly lower than the number of functions that exhibit a prologue, it is expected, as the diversity of prologues is certainly limited in general. Nevertheless, the number of distinct normalized prologues provides a measure of the diversity of function prologues. Since the tools might also consider the bytes before the function starts in their detection, we have included the number of functions with at least one padding byte before the function start in the rightmost column of Table 4.

For comparison, the BAP x64 PE dataset contains 65,733 byte-unique functions, i.e., unique function byte sequences. With normalization, only 18,169 normalized functions (ca. 28%) remain. Similarly, out of 10,979 byte-unique prologues based on `.pdata` section, 1,775 normalized prologues (ca. 16%) remain. Both the absolute numbers and the relative numbers, i.e., the number of normalized functions over the total number of functions, demonstrate fewer duplicates compared to the BAP dataset, indicating that the *FuncPEval* dataset contains more diversity, which is a desirable property for an evaluation dataset.

### 5.2 Evaluation and Tool Comparison

To assess the performance of function start detection tools on PE files, we utilize our newly introduced comprehensive *FuncPEval* dataset to evaluate eight tools, measuring their performance in terms of speed and  $F_1$ -score. The research question is as follows: How do function detection tools perform when predicting function starts in PE programs with diverse code that exhibit different build toolchains? For the tool comparison, we select function start

<sup>7</sup>[http://commondatastorage.googleapis.com/chromium-browser-snapshots/index.html?prefix=Win\\_x64/1069922/](http://commondatastorage.googleapis.com/chromium-browser-snapshots/index.html?prefix=Win_x64/1069922/), chrome.dll has a SHA256 hash value of 55f05fe24ebdf8eb263f75e88c8a71a42fb6240b59340a9abf9671ffe79a4f4a

<sup>8</sup><http://commondatastorage.googleapis.com/chromium-browser-snapshots/index.html?prefix=Win/1069956/>, chrome.dll has a SHA256 hash value of 1ce8b9551709581688a8199a0e0fcb48cfac7fadf3671622ea8e66fbc39151f

<sup>9</sup>See <https://learn.microsoft.com/en-us/windows/win32/debug/pe-format#the-pdata-section>

**Table 4: Properties of the training and the evaluation datasets, showing the number of functions and prologues. Normalization includes blinding immediates as well as call and jump targets.**

| Dataset   | PE sample(s)  | Arch | RVAs    | Functions     |               | Prologues (.pdata) |        |       | Padding instances |
|-----------|---------------|------|---------|---------------|---------------|--------------------|--------|-------|-------------------|
|           |               |      |         | unique        | norm.         | present            | unique | norm. |                   |
| BAP PE    | 68 samples    | x64  | 94,548  | 65,733 (70%)  | 18,169 (19%)  | 74,057             | 10,979 | 1,775 | 84,069            |
| FuncPEval | Chromium v109 | x86  | 548,534 | 541,707 (99%) | 232,781 (42%) |                    |        |       | 377,769           |
| FuncPEval | Chromium v109 | x64  | 542,902 | 536,182 (99%) | 315,745 (58%) | 470,317            | 7,488  | 1,982 | 390,063           |
| FuncPEval | Conti v3      | x86  | 722     | 721 (99%)     | 524 (73%)     |                    |        |       | 310               |
| FuncPEval | Conti v3      | x64  | 662     | 659 (99%)     | 450 (68%)     | 389                | 179    | 122   | 548               |

detection tools listed in Table 1 that support PE and have been published within the past decade (2015-2025). This spans the three learning-based approaches Shin et al. RNN, XDA, and DeepDi [28], the three non-learning-based tools Nucleus [3], SMDA [8, 20], and rev.ng [9], as well as two popular industry tools IDA Pro 7.7 [23] and Ghidra 10.0.4 [1]. For our one output neuron RNN, we train two models separately for x86 and x64, using the full BAP dataset, described in Section 4.1, spanning PE samples with ground truth, compiled and linked with MS Visual Studio versions 2010 to 2013 and using the optimization levels Od, O1, Ox, and O2. Similarly, we use the XDA models described in Section 4.3, trained on 10% of the BAP-64 dataset. For DeepDi, we use the provided model, trained on a mixture of PE files from LLVM, SPEC CPU2006, and SPEC CPU2017. Note that in contrast to the training datasets used by the learning-based tools, the evaluation dataset spans PE samples built with newer (MSVS 2022) or different (LLVM clang) compilers. This approach allows us to work towards evaluating the generalizability of the learning-based methods.

We evaluated the one output neuron RNN, XDA (in the original and our modified version), DeepDi, SMDA, IDA Pro, Ghidra, Nucleus, and rev.ng on the FuncPEval dataset, specifically Chromium and Conti for both x86 and x64. The experiments for the RNN, SMDA, IDA, Ghidra, Nucleus, and rev.ng ran on a server with two 2.20 GHz Intel Xeon Silver 4114 CPUs à 20 threads and 128 GB of RAM. The experiments for XDA and DeepDi ran on a server with two 2.9 GHz AMD EPYC 7542 CPUs à 64 threads, 256 GB of RAM, and one NVIDIA Quadro RTX 8000.

Figure 5 shows the processing time for each tool and sample in seconds, averaged over both the x86 and x64 duration. For Chromium, rev.ng did not finish after 7 days, so we could not obtain results. For Chromium x64, DeepDi is the fastest, completing in seconds, followed by the RNN and Nucleus, both taking minutes. XDA, Ghidra, and IDA ran for hours while SMDA finished after 35 hours.

Concerning F<sub>1</sub>-scores on the x64 Chromium sample, IDA (98.44%) performs best, followed by DeepDi (97%), SMDA (95.54%), and Ghidra (92.48%). Nucleus (88.52%), XDA (86.98%), and the RNN (84.66%) score below 90%. Even though the F<sub>1</sub>-scores of the RNN and XDA are much lower than in Section 4, both still detect a significant number of functions in the evaluation dataset, given the difference between training and evaluation datasets. Overall, the RNN and XDA exhibit higher precision than recall. This reflects that most predicted function starts are indeed function starts according to the ground truth, but a significant number of functions is missed

(in the worst case, up to 33% for the RNN and 20% for XDA). At first, this might indicate that the tools reliably detect function starts they encountered during training while failing to recognize a significant portion of function starts that were not part of the training data. However, the results of Section 5.3 raise doubts whether these tools generalize over code properties observed in function starts.

**Lessons learned.** IDA, DeepDi, SMDA, and Ghidra achieve the best results in function start detection for PE files, with F<sub>1</sub>-scores exceeding 90%. While Nucleus, XDA, and the RNN produce slightly lower results, with F<sub>1</sub>-scores above 80%, they remain valuable tools, particularly due to their relatively fast execution speeds. If precise function detection is the primary focus, IDA is the most suitable tool. However, if execution speed is a critical factor, DeepDi is the preferred choice, as it delivers the second-best F<sub>1</sub>-scores while being by far the fastest.

### 5.3 Randomizing the Padding between Functions

To gain a better understanding of which features are predominantly utilized by the ML-based tools, we analyze the overlap between the detected functions and the training dataset. The RNN detects a total of 439,363 functions in Chromium x64, with 415,810 of them (95%) being true positives. For 353,995 of those functions, we have information about the prologue from the .pdata section. Only 138,201 out of those 353,955 functions (39%) have a normalized prologue that also occurs in the training dataset. Therefore, the RNN detects function starts for which the normalized prologue was not seen during training. We observe the same for XDA. Since we do not have training data for DeepDi, we cannot measure the overlap of normalized function prologues.

These values indicate that the RNN and XDA may not primarily rely on function prologues when detecting function starts. Following Compiler Coding Rule 12 of Intel’s Architecture Optimization Reference Manual, compilers align the first instruction of each function at multiples of 16 bytes [13] and pad the area between the end of the preceding function and the beginning of the next with specific bytes. Typically, the padding is filled with opcode 0xcc, which reflects the mnemonic INT 3, or opcode 0x90, which is a single-byte NOP instruction. Such padding forms a characteristic pattern before a function start. Therefore, we suspect that the RNN and XDA learn that a series of padding bytes is directly followed by a function start, a pitfall referred to as a spurious correlation by Arp et al. [6], as the

| Tool                                | Sample                    | avg.<br>Time | PE x86    |        |          | PE x86-64 |        |          |
|-------------------------------------|---------------------------|--------------|-----------|--------|----------|-----------|--------|----------|
|                                     |                           |              | Precision | Recall | F1-score | Precision | Recall | F1-score |
| RNN<br>INT3-Padding                 | Chromium v109             | 411          | 94.97%    | 67.32% | 78.79%   | 94.64%    | 76.59% | 84.66%   |
|                                     | Chromium v109 RND-Padding | 389          | 47.17%    | 3.23%  | 6.05%    | 62.30%    | 11.20% | 18.99%   |
|                                     | Conti v3                  | 1            | 97.12%    | 88.64% | 92.69%   | 99.06%    | 95.32% | 97.15%   |
|                                     | Conti v3 RND-Padding      | 1            | 91.69%    | 53.46% | 67.54%   | 88.61%    | 27.04% | 41.44%   |
| RNN<br>RND-Padding                  | Chromium v109             | 364          | 1.53%     | 0.65%  | 0.91%    | 41.08%    | 32.37% | 36.21%   |
|                                     | Chromium v109 RND-Padding | 369          | 4.65%     | 2.28%  | 3.06%    | 40.01%    | 35.87% | 37.83%   |
|                                     | Conti v3                  | 1            | 82.81%    | 76.04% | 79.28%   | 92.31%    | 63.44% | 75.20%   |
|                                     | Conti v3 RND-Padding      | 1            | 81.93%    | 76.59% | 79.17%   | 90.81%    | 65.71% | 76.25%   |
| XDA<br>INT3-Padding                 | Chromium v109             | 4872         | -         | -      | -        | 96.30%    | 63.46% | 76.51%   |
|                                     | Chromium v109 RND-Padding | 4869         | -         | -      | -        | 31.80%    | 1.16%  | 2.23%    |
|                                     | Conti v3                  | 6            | -         | -      | -        | 99.63%    | 80.97% | 89.33%   |
|                                     | Conti v3 RND-Padding      | 6            | -         | -      | -        | 84.62%    | 3.32%  | 6.40%    |
| XDA<br>new encoding<br>INT3-Padding | Chromium v109             | 5011         | -         | -      | -        | 94.88%    | 80.29% | 86.98%   |
|                                     | Chromium v109 RND-Padding | 5127         | -         | -      | -        | 40.61%    | 6.70%  | 11.50%   |
|                                     | Conti v3                  | 5            | -         | -      | -        | 97.17%    | 93.20% | 95.14%   |
|                                     | Conti v3 RND-Padding      | 5            | -         | -      | -        | 68.42%    | 27.49% | 39.22%   |
| XDA<br>new encoding<br>RND-Padding  | Chromium v109             | 5013         | -         | -      | -        | 2.81%     | 1.77%  | 2.17%    |
|                                     | Chromium v109 RND-Padding | 5075         | -         | -      | -        | 5.45%     | 1.08%  | 1.80%    |
|                                     | Conti v3                  | 5            | -         | -      | -        | 1.34%     | 1.21%  | 1.27%    |
|                                     | Conti v3 RND-Padding      | 5            | -         | -      | -        | 2.41%     | 0.60%  | 0.97%    |
| DeepDi                              | Chromium v109             | 16           | 96.72%    | 99.81% | 98.24%   | 96.29%    | 97.71% | 97.00%   |
|                                     | Chromium v109 RND-Padding | 16           | 86.68%    | 66.70% | 75.39%   | 73.22%    | 59.60% | 65.71%   |
|                                     | Conti v3                  | 3            | 94.63%    | 97.65% | 96.11%   | 99.84%    | 96.83% | 98.31%   |
|                                     | Conti v3 RND-Padding      | 0.5          | 91.95%    | 94.88% | 93.39%   | 91.40%    | 86.71% | 88.99%   |
| SMDA<br>1.13.21                     | Chromium v109             | 129k         | 95.94%    | 99.86% | 97.86%   | 92.04%    | 99.33% | 95.54%   |
|                                     | Chromium v109 RND-Padding | 138k         | 90.08%    | 96.89% | 93.36%   | 85.30%    | 92.13% | 88.58%   |
|                                     | Conti v3                  | 3            | 90.79%    | 96.95% | 93.77%   | 95.94%    | 99.85% | 97.85%   |
|                                     | Conti v3 RND-Padding      | 3            | 86.84%    | 93.21% | 89.91%   | 89.41%    | 95.62% | 92.41%   |
| IDA Pro<br>7.7                      | Chromium v109             | 32810        | 97.71%    | 99.36% | 98.53%   | 97.11%    | 99.80% | 98.44%   |
|                                     | Chromium v109 RND-Padding | 28415        | 97.67%    | 98.94% | 98.30%   | 88.95%    | 99.80% | 94.06%   |
|                                     | Conti v3                  | 17           | 98.05%    | 97.37% | 97.71%   | 99.85%    | 99.40% | 99.62%   |
|                                     | Conti v3 RND-Padding      | 17           | 97.40%    | 93.49% | 95.41%   | 92.13%    | 99.09% | 95.49%   |
| Ghidra<br>10.0.4                    | Chromium v109             | 24775        | 97.52%    | 98.12% | 97.82%   | 97.04%    | 88.33% | 92.48%   |
|                                     | Chromium v109 RND-Padding | 68973        | 97.17%    | 87.11% | 91.87%   | 97.00%    | 87.98% | 92.27%   |
|                                     | Conti v3                  | 47           | 95.94%    | 91.55% | 93.69%   | 100.00%   | 92.30% | 95.99%   |
|                                     | Conti v3 RND-Padding      | 46           | 95.87%    | 86.70% | 91.05%   | 100.00%   | 85.50% | 92.18%   |
| Nucleus<br>linear sweep             | Chromium v109             | 765          | 97.19%    | 97.13% | 97.16%   | 83.69%    | 93.94% | 88.52%   |
|                                     | Chromium v109 RND-Padding | 633          | 32.70%    | 33.28% | 32.99%   | 27.76%    | 31.97% | 29.71%   |
|                                     | Conti v3                  | 0.5          | 92.19%    | 96.40% | 94.25%   | 94.56%    | 97.13% | 95.83%   |
|                                     | Conti v3 RND-Padding      | 0.5          | 49.94%    | 55.54% | 52.59%   | 22.07%    | 24.17% | 23.07%   |
| Nucleus<br>recursive traversal      | Chromium v109             | 630          | 97.66%    | 90.99% | 94.21%   | 86.24%    | 87.00% | 86.61%   |
|                                     | Chromium v109 RND-Padding | 635          | 42.41%    | 39.48% | 40.89%   | 36.65%    | 37.58% | 37.11%   |
|                                     | Conti v3                  | 0.7          | 92.90%    | 95.98% | 94.41%   | 94.51%    | 96.22% | 95.36%   |
|                                     | Conti v3 RND-Padding      | 0.7          | 60.03%    | 64.68% | 62.27%   | 33.69%    | 38.52% | 35.94%   |
| rev.ng                              | Chromium v109             | -            | -         | -      | -        | -         | -      | -        |
|                                     | Chromium v109 RND-Padding | -            | -         | -      | -        | -         | -      | -        |
|                                     | Conti v3                  | 130          | 88.21%    | 90.17% | 89.18%   | 100.00%   | 78.55% | 87.99%   |
|                                     | Conti v3 RND-Padding      | 128          | 88.09%    | 90.17% | 89.12%   | 100.00%   | 78.55% | 87.99%   |

**Figure 5: Precision, recall, and F<sub>1</sub>-score when predicting function starts in Chromium and Conti samples. Each cell is shaded in red, with the intensity of the red color increasing as the value deviates further from 100%. *INT3-Padding* in tool names indicates training on samples with INT3 instruction as padding between functions, while *RND-Padding* indicates training on samples with random byte values as padding. *RND-padding* in sample names denotes the replacement of compiler-generated padding with random byte values.**

padding bytes are not required for the execution of a binary and can be arbitrarily altered in value. To confirm our hypothesis, we replace the characteristic padding bytes with random byte values in our evaluation dataset and rerun the previous experiment.

*5.3.1 Introducing Random Padding in the FuncPEval Dataset.* We use the FuncPEval dataset described in Section 5.1 and modify the samples to understand the impact of padding bytes on the evaluation. In our original dataset, we only observe padding with opcode

```

1 # iterate over the 20 bytes preceding the function
2 for i in range(1, 20):
3     current_byte =
4         sample_bytes[current_function_start - i]
5     # make sure the current byte does not belong
6     # to another function
7     if belongs_to_function(current_byte):
8         break
9     # make sure the byte is actually a padding
10    # byte
11    if current_byte.value == 0xcc:
12        # replace byte value by random value
13        current_byte.value = random_byte()

```

**Listing 1: Pseudocode of algorithm used to replace padding bytes**

0xcc (mnemonic INT 3 representing a software interrupt). Listing 1 shows the algorithm used to replace the padding bytes. For each function in the samples, we i) collect up to 20 bytes before the beginning of the function, ii) consider only those bytes that do not belong to a preceding function, and iii) replace each padding byte of value 0xcc with a random arbitrary byte value. We select 20 bytes preceding the function to cover any possible padding, which can be up to 15 bytes in length, with an additional margin for tolerance. This modification does not impact the runtime behavior of the samples because the padding is not part of the control flow and is never executed. Therefore, malicious actors could arbitrarily alter the values of the padding bytes as an obfuscation to impede automated analyses. We have observed non-standard padding used by malicious actors in the wild. The winloader<sup>10</sup> malware uses opcode 0xC3 as inter-function padding, which reflects the mnemonic RET. While possibly an artifact of a rare compiler or compiler configuration, or resulting from deliberate manipulation, the intentions for using non-standard padding are unclear in this specific case.

**5.3.2 Results.** Figure 5 shows that the padding significantly impacts the performance of the RNN and XDA. In the case of Chromium x86, the F<sub>1</sub>-score dropped from 78.79% to 6.05% for the RNN. For our modified version of XDA, the F<sub>1</sub>-score dropped from 86.98% to 11.50% for Chromium x64. The modified padding also negatively affects DeepDi, the third machine learning approach, albeit significantly less (drop from 97% to 65.71% in F<sub>1</sub>-score for Chromium x64). We draw two conclusions: First, if a characteristic padding byte pattern is present during training, the RNN and XDA predominantly rely on such a pattern as a delimiter of functions. Second, unlike RNN and XDA, DeepDi is less affected, most likely as it operates on the granularity of machine code instructions instead of raw bytes.

Given a learning-based approach, it may be considered unfair to train on samples with unmodified padding and evaluate against randomized padding. To accommodate, we applied the padding randomization to the samples in the BAP training dataset of the RNN and trained a new model to see if the results improve. The new model (*RND-Padding* in Figure 5) performs overall worse on samples without random padding in comparison to the original model. In the case of Chromium x86, the new model even produces

worse results for the modified sample. Therefore, we conclude it is not straightforward to train an RNN model that can handle both normal and randomized padding effectively, which raises the question of whether the RNN can identify function start patterns beyond padding. We also finetuned XDA using a randomized-padding version of the newly encoded dataset. Overall, the results were significantly worse for both samples with unmodified padding and samples with randomized padding, even though similar results to those in the experiment in Section 4.3 were achieved during validation in training. Consequently, we were unable to train a model for XDA that is robust against randomized padding. Due to the training code of DeepDi not being publicly available, we could not retrain a version of DeepDi.

The randomized padding also negatively affects IDA, Ghidra, and SMDA; however, the impact on their F<sub>1</sub>-scores never exceeds 10 percent points. rev.ng is the only tool that is nearly unaffected by the randomized padding; however, we cannot make a statement regarding its results on Chromium.

Nucleus’ detection of function starts is severely negatively impacted by randomized padding. Both precision and recall are similarly affected. In the case of Chromium x86, the F<sub>1</sub>-score drops from 97.16% to 32.99%. Theoretically, modifying padding bytes should not affect Nucleus’ function start detection, unlike the machine learning tools that have learned with padding during training, since Nucleus operates on the interprocedural control flow graph (ICFG), which does not include padding. Through code review and debugging, we have confirmed that Nucleus is already affected by the randomized padding before creating the ICFG, specifically during disassembling. Nucleus disassembles using linear sweep. The randomized padding bytes are interpreted as instructions by the linear sweep disassembler. This can result in instructions consuming parts of the randomized padding and the beginning of the subsequent function. In this case, at least the function’s first instruction is incorrectly disassembled. The effect is exacerbated by Nucleus detecting that callers point to the middle of an instruction. Nucleus attempts to fix this by shifting the start of the basic block to the beginning of the next instruction. This shift does not resolve the issue, resulting in an incorrect function start being assumed in such cases. Nucleus also provides an experimental recursive traversal disassembling strategy, which shows no significant improvement. While the padding bytes are initially ignored by the recursive strategy, a heuristic<sup>11</sup> later reconsiders them, leading to the same problem as with the linear sweep. This heuristic is intended to improve code coverage by assuming another basic block after the end of the current basic block. Removing this heuristic significantly improves precision but also greatly reduces recall, as the overall code coverage becomes very low.

**Lessons learned.** Function start detection tools are significantly affected by modifications to the compiler-generated padding between functions. When this padding is replaced with random byte values, detection performance deteriorates for the RNN, XDA, DeepDi, and Nucleus. In contrast, IDA, Ghidra, and SMDA are much less impacted. Attempts to retrain

<sup>10</sup>SHA256 hash value of: 72b92683052e0c813890caf7b4f8bfd331a8b2afc324dd545d46138f677178c4

<sup>11</sup><https://bitbucket.org/vusec/nucleus/src/e3ab49db579adbdd8451171e980e9b8f8a546a3c/strategy.cc#lines-149>

the RNN and XDA using samples with randomized padding did not resolve the issue, indicating generalization limitations and leaving it unclear whether these tools can operate effectively on such samples.

## 6 LIMITATIONS & CONCLUSION

We conclude that randomized padding between functions in PE binaries significantly diminishes the effectiveness of the RNN, XDA, and Nucleus. Even training on samples with randomized padding does not resolve the issue for the learning-based methods RNN and XDA, highlighting their limitations in generalizability. The remaining tools are also affected by randomized padding, although to a much lesser degree. Threat actors may evade the affected tools through randomized padding impacting subsequent analysis toolchains, e.g., for malware analysis. Among the learning-based tools, DeepDi is the least affected and, overall, the fastest.

When considering the unmodified version of Chromium x64, IDA (98.44%) performs best, closely followed by DeepDi (97%) and SMDA (95.54%). For large-scale applications, DeepDi likely offers the best combination of F1-score and processing speed.

Finally, by modifying the label encoding, we improve XDA's F<sub>1</sub>-score significantly, resulting in an F<sub>1</sub>-score of 86.98% for Chromium x64 in comparison to 76.51% for the unmodified version.

Since no pre-trained models for x86 were provided for XDA, we decided not to include XDA in our x86 evaluation. Future work could incorporate a newly pre-trained and finetuned version of XDA for x86 in the evaluation. We assume that the results do not differ significantly between x86 and x64. Another limitation is that our dataset only contains one malware family (Conti), using a C/C++ code base. Ideally, it would be extended to also cover malware using different obfuscation techniques, compilers, and toolchains such as Rust, Nim, Go, and corresponding ground truth. However, obtaining such ground truth data is challenging due to missing or partial debug symbols. Finally, while our work focuses on the Windows PE file format, randomized padding likely also impacts the function detection in other executable file formats, such as ELF, given that padding is an architectural recommendation. Future work could investigate the impact of randomized padding on other file formats.

## CODE ARTIFACTS

To foster future research, we publish our source code, data, and other supplementary information online at: <https://github.com/internet-sicherheit/Padding-Matters---Exploring-Function-Detection-in-PE-Files>.

## ACKNOWLEDGMENTS

The authors gratefully acknowledge funding from the *Federal Ministry of Education and Research* (grants 13FH101KB1 and 16KIS1746), *nicos AG*, and *Cyberus Technology GmbH*. The authors thank Jan Fedler for his assistance with debugging Nucleus.

## REFERENCES

- [1] National Security Agency. Ghidra Software Reverse Engineering Framework. <https://ghidra-sre.org/>, 2022.
- [2] Jim Alves-Foss and Jia Song. Function boundary detection in stripped binaries. In *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC '19*, page 84–96, New York, NY, USA, Dec 2019. Association for Computing Machinery.
- [3] Dennis Andriess. Nucleus function detector. <https://bitbucket.org/vusec/nucleus/src/e3ab49db579adbdd8451171e980e9b8f8a546a3c/>, 2018.
- [4] Dennis Andriess, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 583–600, 2016.
- [5] Dennis Andriess, Asia Slowinska, and Herbert Bos. Compiler-Agnostic Function Detection in Binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroSP)*, page 177–189, Apr 2017.
- [6] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. Dos and Don'ts of Machine Learning in Computer Security. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3971–3988, Boston, MA, August 2022. USENIX Association.
- [7] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *23rd USENIX Security Symposium (USENIX Security 14)*, page 845–860, 2014.
- [8] Daniel Johannes Plohmann. *Classification, Characterization, and Contextualization of Windows Malware using Static Behavior and Similarity Analysis*. PhD thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, July 2022.
- [9] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. REV.NG: A Unified Binary Analysis Framework to Recover CFGs and Function Boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 131–141, 2017.
- [10] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. LEMNA: Explaining Deep Learning based Security Applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18*, page 364–379, New York, NY, USA, Oct 2018. Association for Computing Machinery.
- [11] Irfan Ul Haq and Juan Caballero. A survey of binary code similarity. *ACM Computing Surveys (CSUR)*, 54(3):1–38, 2021.
- [12] Hinton. Neural Networks for Machine Learning. [http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture\\_slides\\_lec6.pdf](http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf), 2012.
- [13] Intel. Intel® 64 and IA-32 Architectures Optimization Reference Manual. <https://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf>, 2012.
- [14] AV-TEST-The Independent IT-Security. AV-ATLAS - Malware & PUA. <https://portal.av-atlas.org/malware>, 2024.
- [15] Soomin Kim, Hyungseok Kim, and Sang Kil Cha. Funprobe: Probing functions from binary code through probabilistic analysis. In Satish Chandra, Kelly Blincoe, and Paolo Tonella, editors, *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 1419–1430. ACM, 2023.
- [16] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. <https://arxiv.org/abs/1412.6980>, 2014.
- [17] Hyungjoon Koo, Soyeon Park, and Taesoo Kim. A Look Back on a Function Identification Problem. In *Annual Computer Security Applications Conference, ACSAC*, page 158–168, New York, NY, USA, Dec 2021. Association for Computing Machinery.
- [18] Chengbin Pang, Ruotong Yu, Dongpeng Xu, Eric Koskinen, Georgios Portokalidis, and Jun Xu. Towards optimal use of exception handling information for function detection. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 338–349. IEEE, 2021.
- [19] Kexin Pei, Jonas Guan, David Williams-King, Junfeng Yang, and Suman Jana. XDA: accurate, robust disassembly with transfer learning. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.
- [20] Daniel Plohmann. SMDA. <https://github.com/danielplohmann/smda>, 2024.
- [21] Jing Qiu, Xiaohong Su, and Peijun Ma. Library functions identification in binary code by using graph isomorphism testings. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, page 261–270, Mar 2015.
- [22] Bernardo Quintero, Alex Berry, Ilfak Guilfanov, and Vijay Bolina. Scaling Up Malware Analysis with Gemini 1.5 Flash. <https://cloud.google.com/blog/topics/threat-intelligence/scaling-up-malware-analysis-with-gemini>, July 2024.
- [23] Hex Rays. IDA Pro. <https://hex-rays.com/ida-pro/>, 2022.
- [24] Nathan Rosenblum, Xiaojin Zhu, Barton Miller, and Karen Hunt. Learning to analyze binary computer code. In *Proceedings of the 23rd national conference on Artificial intelligence - Volume 2, AAAI'08*, page 798–804, Chicago, Illinois, Jul 2008. AAAI Press.
- [25] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing Functions in Binaries with Neural Networks. In *24th USENIX Security Symposium (USENIX Security 15)*, page 611–626, 2015.
- [26] Paria Shirani, Lingyu Wang, and Mourad Debbabi. BinShape: Scalable and Robust Binary Library Function Identification Using Function Shape. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware*,

- and Vulnerability Assessment*, Lecture Notes in Computer Science, page 301–324, Cham, 2017. Springer International Publishing.
- [27] vx-underground.org. Conti v3. <https://github.com/vxunderground/MalwareSourceCode/blob/main/Win32/Ransomware/Win32.Conti.c.7z>, 2022.
- [28] Sheng Yu, Yu Qu, Xunchao Hu, and Heng Yin. DeepDi: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2709–2725, Boston, MA, August 2022. USENIX Association.