RustBound: Function Boundary Detection over Rust Stripped Binaries

Ryan Evans, William Hawkins, and Boyang Wang

University of Cincinnati, Cincinnati, OH, USA
evans2ra@mail.uc.edu, hawkinwh@ucmail.uc.edu, boyang.wang@uc.edu

Abstract. Function boundary detection identifies start addresses and end addresses of functions in a binary. It is a critical step in binary analysis and is considered as a challenging task over stripped binaries. While existing studies have shown that it is feasible to efficiently and accurately perform function boundary detection over C stripped binaries, it remains unknown whether these methods will perform well over Rust stripped binaries. In this paper, we experimentally evaluate and compare four methods/tools, including two industry reverse engineering tools (Ghirda and IDA Pro) and two neural-network-based methods, in the context of function boundary detection over Rust binaries. We establish a large-scale dataset consisting of 2,471 Rust binaries (with over 8.69 million functions) across five optimization levels and develop two tools to perform analyses automatically. We derive two major findings based on our experimental results. First, one of the two neural-network-based methods, named XDA, can achieve promising results (e.g., 94.8% precision and 85.5% recall over binaries compiled with O0) and outperform other methods/tools in detecting function boundaries over Rust binaries, except over binaries from Oz optimization. Second, although Ghidra and IDA Pro can accurately detect function starts, they are not effective on precisely distinguishing function ends over Rust binaries.

1 Introduction

Function boundary detection identifies addresses of function starts and function ends over binaries. It is a fundamental component in binary analysis and is critical for downstream tasks, such as function similarity analyses, binary rewriting, and malware detection [8] [15]. However, function boundary detection over *stripped binaries* is considered as a challenging task as function information are no longer available in stripped binaries.

Several studies [7] [18] [9] [5] [4] [16] [20] [8] have proposed to utilize neural networks to address function boundary detection over stripped binaries. These methods can even outperform state-of-the-art reverse engineering tools, including Ghidra [1] and IDA Pro [2]. Despite promising results over binaries compiled from C programs in the current literature, *it remains unknown whether neural-network-based methods would perform well over Rust binaries — binaries compiled from Rust programs.*

Rust is considered as a safer system-level programming language than C by enforcing memory safety and concurrency safety [3] [12]. Rust has become more popular in the past several years, especially for embedded systems [6]. Despite its safety enhancement, some recent research also demonstrate that critical security vulnerabilities can still be found in Rust programs [13] [14] [11]. Therefore, we believe that understanding the capability of binary analysis, including function boundary detection, over Rust binaries is critical and necessary.

In this paper, we leverage four existing methods/tools in the context of function boundary detection and experimentally compare their performance over large-scale stripped Rust binaries. Specifically, we investigate two state-of-the-art reverse engineering tools, including Ghidra and IDA Pro, and two neural-network-based methods, denoted as BiRNN [18] and XDA [16]. Both BiRNN and XDA are originally designed for function detection over C binaries. BiRNN is built upon bi-directional Recursive Neural Networks while XDA is based on masked Language Modeling. Our contributions and observations are summarized as below:

- We establish a large-scale dataset, named RUBIN, which consists of 2,471 Rust binaries (x86-64 in ELF – Executable and Linkable Format) with over 8.69 million functions across five optimizations, including O0, O1, O2, O3, and Oz. The entire dataset (including stripped and non-stripped binaries) is over 66 GBs, where non-stripped binaries are kept for reproducibility and future expansion.
- We develop two open-source tools, named `ripkit` and `cargo_picky`, that can automatically obtain Rust stripped binaries and obtain results of function boundary detection over the four methods we examine. These tools can be utilized to reproduce our results and expand our findings over new datasets/methods over Rust binaries. We believe that both our dataset and the tools are valuable contributions to the research community.
- Our experimental results suggest that (1) XDA can achieve promising results (e.g., 94.8% precision and 85.5% recall over binaries compiled with O0) in function boundary detection and outperform the other 3 methods/tools, except over stripped binaries from Oz optimization; (2) both Ghidra and IDA Pro perform very well on detecting function starts (e.g., $\geq$92.7% precision and $\geq$97.7% recall) but are less promising on identifying function ends (e.g., as low as 55.8% precision and 56.4% recall).

Reproducibility. Our source code and dataset are publicly available at `https://github.com/UCdasec/RustBound`.

2 Background

Rust. Rust is a relatively new programming language which has a similar syntax and performance as C and C++. However, Rust is safer than C and C++ by offering memory safety and concurrency safety. Specifically, Rust catches vulnerabilities, such as memory corruption, race conditions, and data races, at

compile time by leveraging the concept of *ownership* and the *borrower checker*. To compile a Rust program into a binary (either a library or an executable), Rust utilizes a specific compiler named `rustc`. Due to its enhanced safety, it has become more popular among developers and has been recently added to the Linux kernel and Microsoft Windows.

A *crate* is the smallest amount of code that the Rust compiler considers at a time[1]. A crate can be a binary crate or a library crate. A binary crate must have a `main` function and can be compiled to an executable. A library crate does not consist of a `main` function and cannot be compiled to an executable. All the crates we evaluated in this study are binary crates.

Stripped v.s. Non-Stripped. When producing a binary from a program, a compiler has an option (`-s`) to remove debugging and symbol information that are not essential for execution. This process, named *stripping*, is typically done to greatly reduce the binary file size as well as making the binary much more difficult to disassemble. If stripping is applied, a binary generated by a compiler is referred to as a *stripped binary*. Otherwise, it is a *non-stripped binary*. In addition to applying at compiler time, stripping can also be performed independently over a non-stripped binary after it has been compiled with the `strip` command. Besides performance improvement, another major benefit of applying stripping is to make it much more difficult for reverse engineering, either from a benign perspective for better protecting intellectual property or from a malicious perspective for hiding malicious code.

Function Boundary Detection. Given a binary (in essence, a sequence of bytes), function boundary detection is a task for outputting a label to every byte in this sequence, where a label is *function start*, *function end*, or *neither*. More formally speaking, given a sequence of bytes $B = (b_1, ..., b_n)$, function boundary detection F outputs a sequence of labels $L = (l_1, ..., l_n)$ as

$$F(B) \rightarrow L = (l_1, ..., l_n), l_i \in \{\texttt{S}, \texttt{E}, \texttt{N}\} \text{ for } 1 \leq i \leq n \tag{1}$$

where l_i is the label of byte b_i and l_i is either `S` (function start), `E` (function end), or `N` (neither). Function boundary detection often happens concurrently with disassembly or directly after. It is a fundamental component in binary analysis and is critical for downstream tasks, such as function similarity analyses, binary rewriting, and malware detection [8].

Function boundary detection is a trivial task over a non-stripped binary as function names, addresses of function starts, and function lengths are available in the headers of a binary. A debugger, such as `gdb`, can interpret these information easily and locate the addresses of function starts and ends. However, it is considered much more challenging to perform over stripped binaries, which do not consist of the information of function names, addresses of function starts, and function lengths.

Examples. We provide several examples of function starts in Rust and show that the (potential) signatures, i.e., the beginning bytes, of functions vary and are not trivial to distinguish. First, we present three examples of function starts

[1] https://doc.rust-lang.org/book/title-page.html

where the binaries are compiled with O0 optimization. In Listing 1.1, a function starts with a `sub` instruction followed with a `lea` instruction. In Listing 1.2, a function starts with an `or` instruction followed with a `mov` instruction. In Listing 1.3, a function starts with a `mov` instruction.

```
0x71a0 <rpn_reckoner_function>:
sub $0x18, %rsp
lea -0x1(%rsi), %rax
bsr %rax, %rcx
...
```

Listing 1.1: Function Start Example 1 (Assembly O0)

```
6a90 <exa_function>:
or %esi, %edi
mov %edi, -0x4(%rsp)
mov -0x4(%rsp), %eax
...
```

Listing 1.2: Function Start Example 2 (Assembly O0)

```
0x169800 <exa_function>:
mov 0x(%rdi), %ax
ret
cs nopw
...
```

Listing 1.3: Function Start Example 3 (Assembly O0)

It is worth to mention that the signatures of function starts (and ends) are constantly complex and difficult to observe across different optimizations. For instance, the following three function starts in binaries compiled with Oz optimization show various patterns of function starts.

```
0x52ff2 <exa_function>:
push %r14
push %rbx
sub $0x38, %rsp
...
```

Listing 1.4: Function Start Example 4 (Assembly Oz)

```
0x52ff2 <exa_function>:
push %rcx
cmp $0x2, %edi
je 53018
...
```

Listing 1.5: Function Start Example 5 (Assembly Oz)

```
0xe40cb <mgart_function>:
shr %rsi
cmp 0x8(%rdi), %rsi
jae e40d7
...
```

Listing 1.6: Function Start Example 6 (Assembly Oz)

Functions with Padding Bytes. It is typical for a compiler to add padding bytes (0x00s or NOPs) at the end of a function. In this study, *our definition of a function end indicates the address of the last byte of a function excluding padding bytes*. As shown in the following example, a function (`foo`) with function length `0x2a` starts at `0x8970` and ends at `0x8999` (i.e., `0x8970` + `0x2a` - 1). There are several padding bytes starting from `0x899a` but before the start of the next function (main) at `0x89a0`.

```
0x8970 <foo>:
...
0x8995: 48 83 c4 38             add $0x38, %rsp
0x8999: c3                      ret
0x899a: 66 0f 1f 44 00 00       nopw

0x89a0 <main>:
...
```

Listing 1.7: Example of A Function with Padding Bytes (Assembly O0)

Evaluation Metric. By following the evaluation metric from existing studies [7] [18] [9] [5] [4] [16] [20] [8], we examine the performance of a method from three aspects (1) function starts, (2) function ends, and (3) function boundaries, using precision, recall, and F1 score. Precision, recall, and F1 score are defined as below.

$$Pecision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1 = \frac{2 \cdot P \cdot R}{P + R}$$

where TP is true positive, FP is false positive, and FN is false negative.

For function starts, it measures the capability of a method on detecting start addresses of functions in a binary. Let set $S = \{s_1, ..., s_m\}$ be the start addresses of all the m functions in binary B, where s_i is the start address of function f_i in B. Given a set of start addresses $S' = \{s'_1, ..., s'_k\}$ reported by a method over binary B, the true positive, false positive, and false negative of function starts are defined as below:

- TP: the number of start addresses that are in both set S and S', where $TP = |S \cap S'|$.

- FP: the number of start addresses that are in set S' but not in set S, where $FP = |S' - S|$.
- FN: the number of start addresses that are in set S but not in set S', where $FP = |S - S'|$.

TP, FP, and FN of function ends are defined similarly by using the end addresses of functions.

A function boundary is reported correctly if both the function start and function end are correct. In other words, the performance of function boundaries can be aggregated based on the performance of function starts and ends. Specifically, we assume that there are m functions in a binary B, where the function starts and ends can be described with a set of start-end address pairs, $D = \{(s_1, e_1), ..., (s_n, e_m)\}$. Let s_i and e_i be the start address and end address of function f_i respectively. Given a set of start-end address pairs $D' = \{(s'_1, e'_1), ..., (s'_n, e'_k)\}$ reported by a method over binary B, we define the true positive, false positive, and false negative of function boundaries over binary B as below

- TP: the number of start-end address pairs that are both in set D and D', where $TP = |D \cap D'|$.
- FP: the number of start-end address pairs that are in set D' but not in set D, where $FP = |D' - D|$
- FN: the number of start-end address pairs that are in set D but not in set D', where $FN = |D - D'|$.

3 Function Boundary Detection Methods

Existing Reverse Engineering Tools. Popular reverse engineering tools, such as Ghidra and IDA Pro, have historically struggled with disassembling stripped binary files. Traditionally, both tools rely on sophisticated dynamic analysis techniques, heuristics, and complex algorithms to first reconstruct the source code of a given binary file in order to perform program boundary detection. When debugging information are unavailable, it introduces many challenges to reconstruct code and detect function boundaries correctly.

Recently, both Ghidra and IDA Pro have began leveraging signature-based approaches to complement dynamic analysis in function boundary detection for functions in standard libraries. For instance, IDA Pro leverages FLIRT algorithm[2] and Ghidra utilizes Function ID in their signature-based approaches[3] respectively. By searching a large-scale signature database and comparing hash values of bytes in a given binary, these tools can detect function entries and exits more effectively.

Function Boundary Detection using Neural Networks. There are several studies have investigated neural networks in function boundary detection

[2] https://hex-rays.com/products/ida/tech/flirt/in_depth/

[3] https://github.com/NationalSecurityAgency/ghidra/blob/master/Ghidra/Features/FunctionID/src/main/doc/fid.xml

over stripped binaries [7] [18] [9] [5] [4] [16] [20] [8]. However, all the findings are based on binaries from C programs. In this study, we specifically discuss two methods and examine their performance over Rust stripped binaries. One method, referred to as BiRNN [18], is the first study utilizing neural networks for function boundary detection. The other method, referred to as XDA [16], is one of the state-of-the-art methods in disassembly and function boundary detection using neural networks.

Details of BiRNN. Shin et al. [18] first leveraged neural networks to address function boundary detection. Specifically, binaries are pre-processed into 1000-byte sequences, where each byte is encoded into a $\mathbb{R}^{256}$ binary vector with one-hot encoding. These byte sequences are passed to a Bi-directional Recursive Neural Network (BiRNN) to train in order to locate function starts or function ends. Once a BiRNN is trained, it predicts labels over bytes in 1000-byte sequences. It is also worth mentioning that one BiRNN can only report function starts or function ends but not both. When reporting results from both function starts and ends, two BiRNNs will need to be trained separately.

Details of XDA. XDA leverages masked Language Modeling – to address function boundary detection [16]. Specifically, XDA applies a specific implementation of BERT (Bidirectional Encoder Representations from Transformers), referred to as RoBERTa. Bytes in binaries are first pre-processed into 512-byte sequences and each byte is encoded into a $\mathbb{R}^{256}$ binary vector with one-hot encoding.

The neural network model in XDA is trained in two phases, including (1) pre-training phase and (2) fine-tuning phase. In the pre-training phase, the model receives byte sequences as inputs where some bytes are marked as missing (i.e., blanked out intentionally). The model is pre-trained to predict the value of missing bytes. The goal of this pre-training phase is to learn semantics in binaries. In the fine-tuning phase, the model is fine-tuned to transfer the knowledge learned in the pre-training phase to a specific task — function boundary detection. Specifically, each byte is assigned three probabilities, including the probabilities of function start, function end, and neither. The one with the highest probability among the three is the predicted label for the byte. The fine-tuning phase updates the parameters of the model by minimizing discrepancies between predicated labels and ground truth labels. Once the model is fine-tuned, it is utilized to predicate labels over bytes in the testing phase.

4 Our Dataset and Tools

Overview of Our Rust Binary Dataset. To investigate function boundary detection over stripped binaries, especially with neural networks, we need to first build a large-scale dataset of Rust binaries as no public Rust binary datasets are available in the current literature. In this paper, a Rust binary indicates that a binary was compiled from a program written in Rust.

Our dataset, referred to as **RUBIN dataset**, includes a total of 2,471 Rust binaries generated with multiple optimizations, including O0, O1, O2, O3, and

Table 1: Overview of Binaries in RUBIN Dataset.

Optimization	Training Set			
	No. of Binaries	No. of Functions	Size (GB) Non-Stripped	Size (MB) Stripped
O0	100	1,031,919	3.6	369
O1	100	425,596	4.7	235
O2	100	369,387	4.1	205
O3	100	436,519	5.1	232
OZ	100	433,901	2.5	143
Total	500	2,697,322	20.0	1,184
Optimization	Test Set			
	No. of Binaries	No. of Functions	Size (GB) Non-Stripped	Size (MB) Stripped
O0	200	2,574,316	8.8	959
O1	195	891,936	10.0	474
O2	190	737,865	8.7	437
O3	193	644,359	8.2	410
OZ	193	1,218,525	7.3	462
Total	1,971	6,066,801	43.0	2,742

Oz, using compiler `rustc` (version 1.70) provided in `Cargo` — Rust's build system and package manager. Each stripped binary is a x86-64 ELF binary and its associated non-stripped binary is also included in our dataset for generating/validating ground truth labels. The total number of functions in `.text` sections of all the stripped binaries in our dataset is 8,764,123. The size of our entire dataset is over 66 GBs (including all the stripped binaries and non-stripped binaries). We split the entire dataset into two subsets, one for training (if needed) and one for testing. More specific information about our dataset is presented in Table 1.

In addition, we develop two tools, named `ripkit` and `cargo_picky`, to generate our dataset and results automatically in a large scale. These two tools can also be leveraged independently to expand our dataset or create new large-scale datasets for function boundary detection over Rust stripped binaries. Specifically, `cargo_picky` can automatically clone and compile crates with `cargo` using various optimization levels in a large scale. `ripkit` serves as a pre-processing tool to transform bytes in binaries into various formats (especially for neural network approaches). In addition, `ripkit` also serves as a database for maintaining the original binaries and their pre-processed data, profiles byte sequence patterns in large binary datasets, and summarizes experimental results automatically.

How Binaries Are Generated in Our Dataset. The binaries in our dataset are generated by following the steps below.

1. We randomly select and retrieve 300 Rust crates from `crates.io` — the largest Rust crate registry. There are more than 133,000 crates available on `crates.io` as of August 2024. A comprehensive list of all the 300 crates we examine can be found at our GitHub repository along with our dataset.

2. Given compiler `rustc` and compile target `x86-64-unknown-linux-gnu`, we select one optimization level (either O0, O1, O2, O3, or Oz), compile all the 300 crates, and obtain associated non-stripped binaries. Note that there is a small number of crates that cannot be compiled successfully given each optimization level (except O0). In addition, our test dataset excludes binaries that have a `.text` section with less than 1,000 bytes[4]. We repeat this process for every optimization level.
3. Next, we apply `strip` command over the non-stripped binaries we derived in the previous step to obtain stripped binaries.

The stripped binaries are utilized for the evaluation of function boundary detection and their non-stripped binaries are kept to generate ground truth labels.

How Ground Truth Labels Are Generated in Our Dataset. As in previous studies [8] [16], we leverage two existing tools, including `lief`[5] and `pyelftools`[6], to create ground truth labels for bytes in `.text` sections. Specifically, given a non-stripped binary, we first leverage `lief` to extract all the bytes in `.text` section. Next, we utilize `pyelftools` to extract the addresses of all the function starts and function sizes from the headers of this binary. In addition, we derive the address of each function end by adding its function size to the address of its function start. Finally, we label each byte in `.text` section as function start, function end, or neither based on the addresses. As all the bytes in `.text` section do not change after applying stripping, the label for each byte remains the same in the corresponding stripped binary.

In addition to `pyelftools`, other existing tools, such as `odjdump`, can also be leveraged to identify ground truth labels for function boundaries over bytes in a binary [8] [16]. ***It is worth mentioning that different tools could lead to minor discrepancies in ground truth labels***. For instance, given the 200 binaries compiled with O0 in our training data, `odjdump` reports 2,583,853 functions while `pyelftools` reports 2,574,316 functions (i.e., 0.37% less). Given the large number of functions in binaries, it is infeasible for us to manually verify which tool is more accurate than others for generating ground truth labels over our dataset. On the other hand, since the discrepancy is minor and it has a negligible impact to the overall evaluation and comparison among methods, we believe choosing either tool to produce labels is sound and reasonable.

5 Evaluation

In this study, we evaluate and compare the performance of four methods/tools for function boundary detection over stripped binaries compiled from Rust program using our RUBIN dataset. Specifically, we reports results from two state-of-the-art industry tools, including Ghidra and IDA Pro, and two neural-network-based methods, including BiRNN and XDA. For a fair comparison, we compare results

[4] BiRNN does not support a `.text` section with less than 1,000 bytes.
[5] https://lief-project.github.io/
[6] https://github.com/eliben/pyelftools

of the four methods in terms of precision, recall, F-1 score, and running time over our test data in every optimization.

5.1 Experiment Settings

All the analyses and experiments are performed on a Linux desktop running Ubuntu 22.04.3 with Intel i9-14900K CPU, 128 GB Memory, and NVIDIA 4080 GPU.

Experiment Settings in Ghidra and IDA Pro. We leverage command-line tools and APIs to automatically decompile a stripped binary and report the addresses of function starts and ends (or function lengths). Specifically, we leverage `idautils.Function()` and `ida_funcs.calc_func_size()` to extract function starts and lengths in IDA Pro. Similarly, we utilize `FunctionManager`, `getBody()`, and `getNumAddress()` in Ghidra's API to derive function starts and lengths. Since both Ghidra v11.0 and IDA Pro v8.3 do not support compiler `rustc` officially, we do not specific the compiler information but use the default setting during the decompilation process.

Experiment Settings for BiRNN. For BiRNN, we implement it based on the description from [18]. We follow the same pre-processing steps as in [18]. Specifically, given bytes from the `.text` section of a stripped binary, we divide the bytes into byte sequences with no overlaps, where each byte sequence has 1,000 consecutive bytes. If a byte sequence does not have 1,000 bytes (e.g., towards the end of the `.text` section), we discard it. We randomly select a total number of 1,000 byte sequences from binaries in the training data of each optimization to train a BiRNN. Each byte in a byte sequence is then encoded into a R^{256} binary vector with one-hot-encoding before passing it as an input to a BiRNN.

The BiRNN model consists of 1 bi-directional RNN layer with 16 RNN hidden units, a single linear layer, and an output layer with `sigmoid` as the activation function. We choose a learning rate of 0.0005 and Binary Cross-Entropy (BCE) as the loss function. The model includes a total of 8,796 parameters.

When reporting the results of BiRNN for each optimization, we obtain all the 1000-byte sequences from the stripped binaries reserved for testing and pass these byte sequences to a BiRNN to obtain the output on each byte. Similar as in [18], sequences with less than 1,000 bytes cannot report outputs and are excluded from results. It is also worth to mention that one BiRNN can only report function starts or function ends but not both. *When reporting results from both function starts and ends, two BiRNNs will need to be trained separately.*

Experiment Settings for XDA. We follow the same pre-processing steps as in [8] and leverage their source code[7]. Specifically, given one optimization, we first randomly select 50 binaries as pre-training data and another 50 binaries as fine-tuning data. These binaries are from RUBIN training set. Second, we extract and form one super byte sequence by concatenating all the bytes from `.text` sections of the 50 binaries in pre-training data. Next, we divide the super sequence into 512-byte sequences and pass these sequences to the model for

[7] https://github.com/CUMLSec/XDA

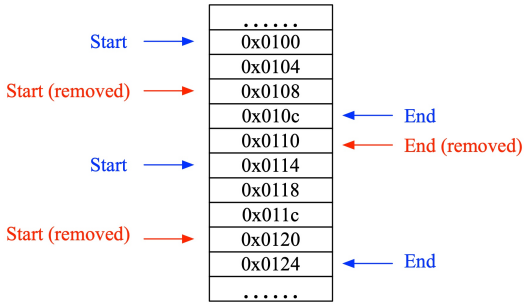


Fig. 1: Example of deriving start-end address pairs based on reported function starts and function ends (for neural-network-based methods only). Given four function starts and three function ends, two start-end address pairs, (0x0100, 0x010c) and (0x0114, 0x0124), are assembled in this example.

pre-training. Masking is randomly applied to 20% of the bytes during the pre-training by the original implementation of XDA. In the fine-tuning phase, a super sequence is formed from the 50 binaries in the fine-tuning data and is divided into 512-byte sequences. In addition, labels are attached to all the bytes. All the 512-byte sequences with labels are passed to the model for fine-tuning. We pre-train each model for 300 epochs and fine-tune each model for 20 epochs.

XDA leverages a transformer-based model, utilizing multi-headed self attention to increase the models attention to long range dependencies. In total, XDA includes 12 self-attention layers, each with 12 self-attention heads. These self-attention layers use the GeLu activation function. while the fine-tuning layers use the tanh function. The encoding layer outputs an embedded with 786 dimensions. The final decoder layer has 2 feed forward networks stack one top of it, one of which is used to predict mask bytes, the other is used to fine-tune to predict the function boundary labels.

When reporting the results of XDA for each optimization, we obtain all the 512-byte sequences from the stripped binaries reserved for testing and pass these byte sequences to XDA to obtain the output on each byte. Similar as in [8], sequences with less than 512 bytes cannot report outputs and are excluded from results. *A single XDA model can reports results in both function starts and ends.*

Reporting Results on Function Bounds. When we report the results on function bounds from Ghidra or IDA, it is straightforward as each tool reports a start address and the length for every function in a binary. This information can be easily transform to the end address of a function as well as a start-end address pair. In addition, there are no overlaps between start-end address pairs reported from Ghidar or IDA. However, for neural-network-based methods, including BiRNN and XDA, there could be cases that a method reports multiple function starts before it identifies the next function end or a function start happens at a lower address than the function end of a previous function. Therefore,

Table 2: The number of start 2-grams (first two bytes for function starts) and the number of end 2-grams (last two bytes for function ends) in RUBIN Dataset (training data only).

	No. of Binaries	No. of Functions	No. of Start 2-Grams	No. of Start 2-Grams (Elsewhere)	No. of End 2-Grams	No. of End 2-Grams (Elsewhere)
O0	100	1,031,919	212	212	27	26
O1	100	425,596	406	406	37	36
O2	100	369,387	394	394	26	25
O3	100	436,519	473	473	36	35
OZ	100	433,901	790	790	143	142

for neural-network-based methods, we perform additional post processing based on the results of function starts and function ends to obtain start-end address pairs such that the start address of a function is the first start address that does not overlap with a previous function. Additional start addresses before the next end address are also removed. An example of post-processing is presented in Fig. 1.

5.2 Experiments

Experiment 1: Understanding the Variety of Function Start and Ends. We first run analyses over our dataset to show the variety of bytes associated with function starts and ends and why function boundary detection is non-trivial. Specifically, we show the number of *start 2-grams* (the first two bytes of a function) and the number of *start 2-grams elsewhere* (the first two bytes of a function but are also found at non-function starts) for every optimization level in Table 2. For instance, given 425,596 functions across 100 binaries from O1 optimization, there are 406 unique start 2-grams. However, all of these 2-grams can also be found at other addresses which are not function starts. This indicates that leveraging these 2-grams as trivial signatures for detecting function starts is not sufficient. More comprehensive signatures should be considered to tackle the problem. We have similar observations from function ends as well when we compare the number of *end 2-grams* (the last two bytes of a function) and the number of *end 2-grams elsewhere* (the last two bytes of a function but are also found at non-function ends) in each optimization.

Experiment 2: Comparison among Different Methods. We report and compare the precision, recall, and F1 score of each method, including Ghirda, IDA Pro, BiRNN, and XDA, over stripped binaries from each optimization level. The detailed results are summarized in Table 3. Overall, we have two major observations.

- ***Observation 1.1:*** *XDA* performs the best in detecting function starts across all the five optimizations with over 99.8% precision and 99.8% recall. In addition, it derives the best results in identifying function ends in

Table 3: Comparison of the four methods/tools (cells highlighted with green are the highest within each optimization).

Train (if needed) & Test	Methods	Function Starts			Function Ends			Function Bounds		
		PR	RE	F1	PR	RE	F1	PR	RE	F1
O0	Ghidra	0.996	0.996	0.996	0.771	0.771	0.771	0.771	0.771	0.771
	IDA Pro	0.992	0.999	0.995	0.779	0.779	0.779	0.779	0.779	0.779
	BiRNN	0.999	0.876	0.934	0.999	0.949	0.973	0.958	0.804	0.875
	XDA	0.999	0.999	0.999	0.999	0.901	0.948	0.948	0.855	0.899
O1	Ghidra	0.998	0.997	0.998	0.631	0.632	0.632	0.631	0.631	0.631
	IDA Pro	0.975	0.986	0.981	0.564	0.564	0.564	0.564	0.564	0.564
	BiRNN	0.997	0.784	0.878	0.997	0.823	0.902	0.879	0.635	0.737
	XDA	0.999	0.999	0.999	0.999	0.883	0.938	0.931	0.823	0.874
O2	Ghidra	0.940	0.941	0.940	0.574	0.574	0.574	0.574	0.574	0.574
	IDA Pro	0.972	0.984	0.978	0.575	0.576	0.575	0.570	0.571	0.571
	BiRNN	0.991	0.841	0.910	0.987	0.881	0.931	0.906	0.709	0.795
	XDA	0.999	0.999	0.999	0.998	0.932	0.964	0.939	0.875	0.906
O3	Ghidra	0.927	0.932	0.929	0.558	0.561	0.559	0.557	0.561	0.559
	IDA Pro	0.969	0.985	0.977	0.570	0.571	0.571	0.570	0.571	0.571
	BiRNN	0.995	0.827	0.903	0.992	0.821	0.899	0.876	0.660	0.752
	XDA	0.998	0.999	0.999	0.998	0.927	0.961	0.939	0.871	0.904
Oz	Ghidra	0.998	0.998	0.998	0.739	0.739	0.739	0.739	0.739	0.739
	IDA Pro	0.979	0.977	0.978	0.653	0.653	0.653	0.654	0.654	0.654
	BiRNN	0.989	0.761	0.860	0.974	0.832	0.898	0.637	0.401	0.492
	XDA	0.998	0.999	0.999	0.999	0.184	0.311	0.952	0.176	0.296

O1, O2, and O3. On the other hand, BiRNN performs the best in O0 and Ghidra outperforms other methods in Oz in terms of detecting function ends.

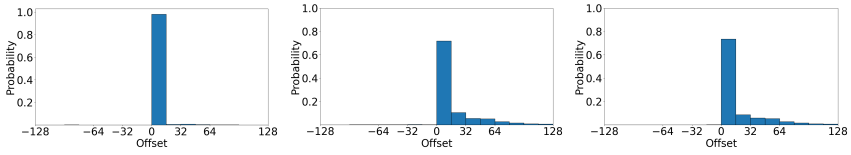
– ***Observation 1.2:*** Ghidra and IDA Pro offers promising performance with high precision and recall in function starts but performs much worse in detecting function ends than XDA (except for binaries with Oz optimization). For instance, IDA Pro achieves 99.2% precision and 99.9% recall in detecting function starts but only 77.9% precision and 77.9% recall in identifying function ends in O1.

In addition, we also present detailed results from Ghidra and IDA Pro to provide more insights. First, we present true positives, false positives, and false negatives of these two methods on function bounds in Table 4 to interpret why the precision and recall of each tool is almost the same in Table 3. This is because the number of functions reported by each tool is very close to the ground truth, which leads to only a minor difference between false positives and false negatives.

Second, we also show to what degree IDA Pro (or Ghidra) fails to detect function ends. Specifically, we report the PDF (Probability Dense Function) of address offsets, where each address offset is the offset between a ground-truth function end and an inferred function end from IDA Pro (or Ghidra). We define an address offset (for function ends) as below. Given a set of (ground-truth) end addresses $E = \{e_1, ..., e_m\}$ and a set of (inferred) end addresses $E' = \{e'_1, ..., e'_k\}$,

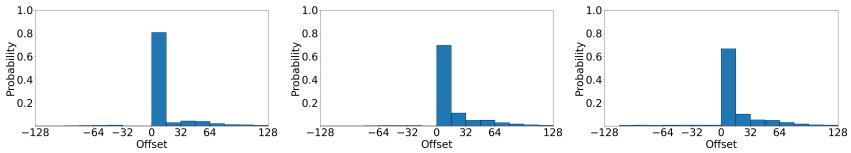
Table 4: True Positive, False Positive, and False Negative of Function Bounds (Ghidra and IDA Pro from Table 3).

	Methods	TP	FP	FN	PR	RE
O0	Ghidra	1,983,879	591,009	590,437	0.771	0.771
	IDA Pro	2,004,180	568,008	570,136	0.779	0.779
O1	Ghidra	563,513	328,959	328,618	0.631	0.632
	IDA Pro	503,400	389,207	388,731	0.564	0.564
O2	Ghidra	423,886	314,675	314,169	0.574	0.564
	IDA Pro	367,857	277,053	276,695	0.570	0.571
O3	Ghidra	361,671	287,277	282,881	0.557	0.561
	IDA Pro	367,857	277,053	276,695	0.570	0.571
Oz	Ghidra	847,384	299,257	298,886	0.739	0.739
	IDA Pro	748,798	398,408	397,472	0.654	0.654



(a) O0 optimization (b) O1 optimization (c) O2 optimization

Fig. 2: The PDFs of address offsets (distance) of IDA Pro on function ends



(a) O0 optimization (b) O1 optimization (c) O2 optimization

Fig. 3: The PDFs of address offsets (distance) of Ghidra on function ends

we obtain a set of address offsets $O = \{o_1, ..., o_k\}$, where

$$o_i = e'_i - e_j, \quad s.t. \quad j = \underset{k \in [1,m]}{\arg\min} |e'_i - e_k|$$

for $1 \leq i \leq m$. As we can see from Fig. 2, although the precision and recall of IDA Pro on function ends are less promising, the address offsets are small, which suggests that the inferred addresses are not far from the ground truth in general. We also have similar observations from Ghidra on function ends as shown in Fig. 3.

While XDA performs the best among the four methods/tools, we would like to acknowledge that XDA requires significant amount of training time (more specifically, pre-training time and fine-tuning time). For instance, it takes about 144 hours to complete the pre-training phase (96 hours) and fine-tuning phase

Table 5: Comparison of the Four Methods/Tools in Training Time and Test Time.

	Ghirda	IDA Pro	BiRNN	XDA
Training Time (hours)	NA	NA	0.03	144
Test Time (KBs/sec)	2.6	4.2	14.13	9.99

(48 hours) per XDA model with our GPU machine. In terms of test time, the four methods are all efficient and report results in large-scales within reasonable amounts of time as shown in Table 5.

Experiment 3: Performance of XDA (Cross-Optimization Scenario). One common limitation of neural-network-based methods is that the performance of a model can drop significantly when there are domain shifts between training data and test data. In binary analysis, a common factor that could lead to domain shifts is compiler optimization [17] [19]. To examine whether XDA is robust again domain shifts due to compiler optimization, we measure the performance of XDA in cross-optimization scenarios. Specifically, we leverage a trained model using binaries from one optimization (e.g., O0) in the previous experiment but test the model with binaries from a different optimization (e.g., O1).

As shown in Table 6, when a trained neural network is tested over binaries with lower optimization levels, such as O0 or O1, XDA still performs well (with slightly performance drops) when training binaries and test binaries are compiled with different optimizations. On the other hand, when a trained neural network is tested over binaries with higher optimization levels, such as O2, O3 or Oz, but the neural network is trained based on O0 or O1, noticeable performance drops (e.g., more than 10% drops) can be observed.

Experiment 4: Performance of XDA (Multi-Optimization Training) To overcome the limitation above, one typical approach is to train a neural network with data from multiple domains, more specifically, bytes from multiple optimizations, such that the neural network is able to generalize, at least for each optimization that is considered during the training. Specifically, we pre-train a unified model of XDA by utilizing bytes from 50 binaries from all the 5 optimizations (including O0, O1, O2, O3, O4, and O5) with 50 binaries per optimization. Next, we fine-tune this model with bytes from 100 total binaries from all the 5 optimizations (20 binaries per optimization). One the model is fined tuned, we report the performance of the model on the test data from each optimization.

We observe that training a single neural network with binaries from multiple optimizations can effectively overcome domain shifts and offer high precision, recall and F1 score over binaries from all optimization except for Oz optimization. The performance of this unified model is almost the same as training each optimization and test each optimization separately shown in Table 3. However, training one unified neural network obviously can reduce computational overhead rather than training five neural networks separately.

Table 6: Performance of XDA in cross-optimization scenarios (cells highlighted with green are results from same-optimization for easy comparison; cells highlighted with red are results with more than 10% drops than same-optimization scenarios.)

		Function Starts			Function Ends			Function Bounds		
Train	Test	PR	RE	F1	PR	RE	F1	PR	RE	F1
O0	O0	0.999	0.999	0.999	0.999	0.901	0.948	0.948	0.855	0.899
O1	O0	0.997	0.985	0.991	0.999	0.897	0.946	0.944	0.843	0.891
O2	O0	0.996	0.985	0.990	0.992	0.898	0.942	0.938	0.841	0.887
O3	O0	0.994	0.989	0.991	0.998	0.901	0.947	0.945	0.850	0.895
Oz	O0	0.989	0.986	0.988	0.999	0.896	0.945	0.943	0.842	0.890
O0	O1	0.989	0.989	0.989	0.998	0.883	0.937	0.929	0.820	0.871
O1	O1	0.999	0.999	0.999	0.999	0.883	0.938	0.931	0.823	0.874
O2	O1	0.999	0.995	0.997	0.999	0.838	0.938	0.999	0.883	0.938
O3	O1	0.999	0.995	0.997	0.999	0.883	0.938	0.931	0.822	0.874
Oz	O1	0.996	0.999	0.997	0.999	0.883	0.938	0.932	0.823	0.874
O0	O2	0.977	0.995	0.986	0.976	0.931	0.953	0.909	0.859	0.883
O1	O2	0.970	0.234	0.378	0.998	0.216	0.355	0.941	0.204	0.335
O2	O2	0.999	0.999	0.999	0.998	0.932	0.964	0.939	0.875	0.906
O3	O2	0.998	0.999	0.999	0.999	0.932	0.964	0.999	0.932	0.905
Oz	O2	0.993	0.999	0.996	0.994	0.932	0.962	0.933	0.872	0.902
O0	O3	0.977	0.995	0.986	0.974	0.926	0.950	0.909	0.856	0.882
O1	O3	0.304	0.004	0.008	0.604	0.001	0.001	0.221	0.000	0.000
O2	O3	0.998	0.999	0.998	0.999	0.927	0.961	0.939	0.871	0.903
O3	O3	0.998	0.999	0.999	0.998	0.927	0.961	0.939	0.871	0.904
Oz	O3	0.993	0.999	0.996	0.994	0.927	0.959	0.933	0.868	0.900
O0	Oz	0.988	0.808	0.889	0.993	0.185	0.312	0.940	0.175	0.294
O1	Oz	0.766	0.014	0.027	0.556	0.000	0.001	0.389	0.000	0.000
O2	Oz	0.994	0.832	0.906	0.999	0.186	0.314	0.943	0.176	0.296
O3	Oz	0.993	0.885	0.936	0.998	0.213	0.352	0.840	0.214	0.295
Oz	Oz	0.998	0.999	0.999	0.999	0.184	0.311	0.952	0.176	0.296

6 Related Work

Function Boundary Detection over C Binaries. Bao et al. [7] designed a function boundary detection boundary method, named ByteWeight. It learns signatures of function starts based on weighted prefix tree and identify function starts by matching binary segments with signatures. Andriesse et al. [5] propose Nucleus, which can detection functions based on control flow graphs and is compiler-agnostic. Alves-Foss and Song [4] developed a function boundary detection method and integrated it into Jima, a tool suite for binary vulnerability analysis and repair. This method leverages explicit calls and jumps as indicators of function starts. It does not require extensive training time compared to neural-network-based approaches. Bundt et al. [8] investigated black-box attacks on neural-network-based binary function detection. Specifically, the authors con-

Table 7: Performance of XDA (Multi-Optimization Training with O0, O1, O2, O3 and Oz).

	Function Starts			Function Ends			Function Bounds		
Test	PR	RE	F1	PR	RE	F1	PR	RE	F1
O0	0.998	0.999	0.999	0.999	0.901	0.948	0.948	0.854	0.899
O1	0.996	0.998	0.997	0.999	0.883	0.938	0.932	0.822	0.874
O2	0.997	0.998	0.997	0.998	0.932	0.964	0.937	0.874	0.905
O3	0.997	0.999	0.998	0.998	0.927	0.961	0.938	0.870	0.902
Oz	0.994	0.997	0.996	0.999	0.184	0.311	0.952	0.176	0.296

sider inadvertent attacks caused by compiler options and adversarial attacks by instruction rewriting (e.g., replacing NOPs with jumps or mov instructions). Yu et al. [20] proposed a method, named DeepDi, which leverages graph neural networks to capture instruction relations. DeepDi is primarily used for disassembly. In addition, it also offers capabilities to identify function starts using heuristics. *All these existing studies report experimental results over C stripped binaries.*

Vulnerabilities in Rust. Li et al. [13] designed a method to detect bugs in Rust programs using static analysis. Liu et al. [14] proposed XRust, a method that changes unsafe Rust code into safe Rust code by leveraging a novel heap allocator. Felix et al. [11] developed a proof-of-concept to show that it is feasible to force a bounds-checked Rust array variable access to read any byte in the memory.

7 Discussions and Future Work

While we examine the performance of four methods/tools over different optimization levels, we did not examine the impacts of the version of the compiler (`rustc`) or compiler flags. Recent work in [8] shows that binaries produced by various compiler flags can lead to domain shifts and could affect the performance of boundary detection over C binaries. These impacts over Rust binaries remain unknown and will be interesting to explore in future research. Whether it is feasible for an attacker to modify Rust stripped binaries and force a neural-network-based method to predict incorrectly on function starts and function ends remains open. Techniques, such as binary rewriting [10], can be examined to address this problem. One of the key challenges would be how to modify Rust binaries automatically in a large scale without affecting functionalities of binaries. It would also be interesting to explore the performance of function boundary detection over Rust binaries with other standard instruction set architectures, such as RISC-V, in future work.

GhidRust[8] is a relatively new open-source tool, which is a Ghidra extension specifically for analyzing Rust binaries. It can decide whether a binary is a Rust binary and report functions in a binary based on signatures using Ghidra's

[8] https://github.com/DMaroo/GhidRust

Function ID. Specifically, it creates a Function ID database for Rust's `libstd` on x86-64 for Rust version 1.58.0 and perform function detection (including function starts, function size, and function names)[9]. Unfortunately, no comprehensive analyses are reported regarding the accuracy of GhidRust's function detection. While the development of this tool has been paused, it will be still interesting to examine the detection performance of this tool over our dataset RUBIN in our future work.

8 Conclusion

We investigate the problem of function boundary detection over Rust stripped binaries. Our experimental results show that a neural-network-based method can achieve very high precision, recall and F1 score and outperform start-of-the-art industry reverse engineering tools in function boundary detection. The neural-network-based method can also render outstanding performance across different optimization levels when it is trained with binaries from multiple optimization levels. Moreover, we develop two automatic tools and one large-scale dataset that can be utilized by the research community to expand and extend research findings on function boundary detection over Rust binaries.

Acknowledgement

The authors thank the anonymous shepherd and reviewers for their comments and suggestions. This work was partially supported by National Science Foundation (CNS-2150086) and UC (University of Cincinnati) Undergraduate Research Fellowship.

References

1. https://ghidra-sre.org/
2. Ida pro, https://hex-rays.com/ida-pro/
3. Rust, https://www.rust-lang.org/
4. Alves-Foss, J., Song, J.: Function Boundary Detection in Stripped Binaries. In: Proc. of ACSAC'19 (2019)
5. Andriesse, D., Slowinska, A., Bos, H.: Compiler-Agnostic Function Detection in Binaries. In: Proc. of Euro S&P'17 (2017)
6. Ayers, H., Laufer, E., Mure, P., Park, J., Rodelo, E., Rossman, T., Pronin, A., Levis, P., Why, J.V.: Tighten Rust's Belt: Shrinking Embedded Rust Binaries. In: Proc. of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'22) (2022)
7. Bao, T., Burket, J., Woo, M., Turner, R., Brumley, D.: ByteWeight: Learning to Recognize Functions in Binary Code. In: Proc. of USENIX Security'14 (2014)
8. Bundt, J., Davinroy, M., Agadakos, I., Oprea, A., Robertson, W.: Black-box Attacks Against Neural Binary Function Detection. In: Proc. of RAID'23 (2023)

[9] https://github.com/DMaroo/GhidRust/blob/master/media/report.pdf

9. Chua, Z.L., Shen, S., Saxena, P., Liang, Z.: Neural Nets Can Learn Function Type Signatures From Binaries. In: Proc. of USENIX Security'17 (2017)
10. Duck, G.J., Gao, X., Roychoudhury, A.: Binary Rewriting without Control Flow Recovery. In: Proc. of ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'20) (2020)
11. Felix, C., Benti, D., Austin, T.: Spectre v1 proof-of-concept attack in the rust language, https://github.com/toddmaustin/spectre-rust
12. House, T.W.: Black to the building blocks: A path toward secure and measureable software, https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf
13. Li, Z., Wang, J., Sun, M., Lui, J.C.: MirChecker: Detecting Bugs in Rust Programs via Static Analysis. In: Proc. of ACM CCS'21 (2021)
14. Liu, P., Zhao, G., Huang, J.: Securing Unsafe Rust Programs with XRust. In: Proc. of IEEE/ACM ICSE'20 (2020)
15. Pang, C., Yu, R., Chen, Y., Koskinen, E., Portokalidis, G., Mao, B., Xun, J.: SoK: All You Ever Wanted to Know About X86/x64 Binary Disassembly But Were Afraid to Ask. In: Proc. of IEEE S&P'21 (2021)
16. Pei, K., Guan, J., Williams-King, D., Yang, J., Jana, S.: XDA: Accurate, Robust Disassembly with Transfer Learning. In: Proc. of NDSS'21 (2021)
17. Ren, X., Ho, M., Ming, J., Lei, Y., Li, L.: Unleashing the Hidden Power of Compiler Optimization on Binary Code Difference: An Empirical Study. In: Proc. of PLDI'21 (2021)
18. Shin, E., Song, D., Moazzezi, R.: Recognizing Functions in Dinaries with Neural Networks. In: Proc. of USENIX Security'15 (2015)
19. Wang, C., Ninan, M., Reilly, S., Ward, J., Hawkins, W., Wang, B., Emmert, J.M.: Portability of Deep-Learning Side-Channel Attacks against Software discrepancies. In: Proc. ACM WiSec'23 (2023)
20. Yu, S., Qu, Y., Hu, X., Yin, H.: DeepDi: Learning a Relational Graph Convolutional Network Model on Instructions for Fast and Accurate Disassembly. In: Proc. of USENIX Security'22 (2022)