Code Pointer Network for Binary Function Scope Identification

Van Nguyen*, Trung Le*, Tue Le†, Khanh Nguyen†, Olivier de Vel‡, Paul Montague‡ and Dinh Phung*
*Faculty of Information Technology, Monash University, Australia
†AI Research Lab, Trusting Social, Australia
‡Defence Science and Technology Group, Australia

***Abstract*—Function identification is a preliminary step in binary analysis for many extensive applications from malware detection, common vulnerability detection and binary instrumentation to name a few. In this paper, we propose the Code Pointer Network that leverages the underlying idea of a pointer network to efficiently and effectively tackle function scope identification – the hardest and most crucial task in function identification. We establish extensive experiments to compare our proposed method with the deep learning based baseline. Experimental results demonstrate that our proposed method significantly outperforms the state-of-the-art baseline in terms of both predictive performance and running time.**

***Index Terms*—Deep Learning, Cyber Security, Function Scope Identification.**

I. Introduction

In computer security, we often face the situation where source code is not available or impossible to access and only binaries are accessible. In these situations, binary analysis is an essential tool enabling many extensive applications such as malware detection [1], common vulnerability detection [2]. Function identification is usually the first step in many binary analysis methods. This aims to identify function scopes in a binary and can contribute to a wide variety of application domains including searching for vulnerabilities [3], binary instrumentation [4], binary protection structures with Control-Flow Integrity [5], and binary software vulnerability detection [6], [7]. One of the most challenging problems in both binary analysis and function identification is how to tackle the scarcity of high-level semantic structures in binaries which might originate from compilers during the process of compilation.

There have been many effective approaches proposed for solving the function identification problem from simple heuristics solutions to more complicated methods employing machine learning or deep learning techniques. In an early work of the function identification problem, [8] pointed out that the task of function start identification is trivially solved for regular binaries. However, later research of [2] and [9] revealed that this task is non-trivial and complex in some specific cases where it is too difficult for heuristics-based methods to figure out all function boundaries. ByteWeight [10] is a machine learning based method for function identification. It learns signatures for function starts using a weighted prefix tree, and recognizes function starts by matching binary fragments with the signatures. Each node in the tree corresponds to either a byte or an instruction, with the path from the root node to any given node representing a possible sequence of bytes or instructions. Although ByteWeight significantly outperformed disassembler approaches such as IDA Pro, Dyninst [11], BAP [12], and the CMU Binary Analysis Platform, it is not scalable enough for even medium-sized datasets. In particular, it took about 587 compute-hours for training a dataset of 2,064 binaries [13]. Recently, [14] proposed a new solution for function identification, which is based on Control Flow Graph analysis. This method was comparable with ByteWeight in terms of predictive performance while requiring less computational time.

The study in [13] is the first work that applied a deep learning technique for the function identification problem. In particular, a Bidirectional Recurrent Neural Network (Bidirectional RNN) was used to identify whether a byte is a start point (or end point) of a function or not. This method was proven to outperform ByteWeight while requiring much less training time. However, this method cannot address the function scope identification problem, the toughest and most essential sub problem in function identification, wherein the scope (i.e., the indexes or addresses of all machine instructions in a function) of each function must be specified. The only way to fulfill this task using this method is to first pair corresponding function starts and function ends and make assumption that the scope inside each pair comprises a function. In addition, if both start points and end points are simultaneously necessary, two separate Bidirectional RNNs must be trained independently and this certainly cannot exploit the semantic relationship among start points, end points, and other machine instructions in a function.

In this paper, we propose a method named the Code Pointer Network (CPN) that employs the idea of a pointer network [15] in the specific context of function identification. Our proposed CPN includes one encoder and one decoder. The encoder takes the sequence of all machine instructions in a binary while the decoder reads out function scopes in the given binary. In addition, unlike the work of [13], our proposed CPN can directly address the function scope identification task, hence it inherently offers the solutions for other simpler tasks including function start, function end, and function boundary

Acknowledgement: This research was supported under the Defence Science and Technology Group's Next Generation Technologies Program.

identifications. We establish extensive experiments to compare our proposed Code Pointer Network with the Bidirectional RNN proposed in [13] on 120,000 binaries compiled from 120,000 C/C++ programs generated by Csmith [16] – a famous tool for generating codes using for the purpose of testing compilers. The experimental results shows that our proposed method significantly outperforms the baseline on function start, function end and function scope identification tasks, especially for the hardest task of function scope identification. Regarding the amount of time taken for training, our proposed CPN is three times faster than baseline (i.e., around two hours in comparison with six hours) using the same number of iterations (i.e., 40,000 iterations).

II. Related Work

In this section, we introduce work related to ours. We firstly present the pointer network and its applications and then move to discuss work in function identification.

A pointer network [15] is a seq2seq model that deals with the case when the target dictionary is varied and non-fixed. The underlying idea of a pointer network is to employ a content-based attention mechanism to form a discrete distribution over inputs. Pointer networks have been applied to several real-world problems including finding convex hulls, Delaunay triangulation, the traveling salesman problem [15], sorting an array [17], and natural language processing [18].

Function identification is not a real problem when working with source codes [19], [20]. However, this is a preliminary step in binary analysis. Besides the works that used heuristics, several machine learning works have attempted to solve this problem. The seminal work of [21] modeled function start identification as a Conditional Random Field (CRF) in which binary offsets and a number of selected idioms (patterns) appear in the CRF. Since the inference on a CRF is very expensive, the computational complexity of this work is very high. [10] used a weighted prefix tree to learn signatures for function starts , and then recognize function starts by matching binary fragments with the signatures. However, because of its high computational complexity, this work is not suitable for large-scale data sources. Recent methods [13] and [22] proposed to employ a Bidirectional RNN for function start (or function end) and function scope identification respectively. However, those work cannot be applied directly to or cannot address all cases (e.g., in the case, there exist functions nested in a function) of the toughest and most essential problem – the function scope identification problem. In 2017, Andriesse et al. [14] based on Control Flow Graph analysis to propose a new solution for function identification.

III. The Function Identification Problem

This section addresses the function identification problem. We begin with definitions of the sub problems in the function identification problem, followed by a typical example of source code generated by Csmith and its binaries compiled under optimization levels O1 and Ox. Finally, we discuss on the challenges in this task.

A. Problem Definitions

Given a binary program P, our task is to identify the necessary information in its n functions $\{f_1, ..., f_n\}$ which is initially unknown. According to the nature of information we need from $\{f_1, ..., f_n\}$, we can categorize the task of function identification into the following such problems.

Function start identification: In this problem, we need to specify the set S $= \{s_1, ..., s_n\}$ which contains the first machine instructions of the corresponding functions in $\{f_1, ..., f_n\}$.

Function end identification: In this problem, we need to identify the set E $= \{e_1, ..., e_n\}$ which contains the end machine instructions of the corresponding functions in $\{f_1, ..., f_n\}$.

Function boundary identification: The function boundary identification problem is harder than the function start and function end identification problem. In this problem, we have to point out the set of (start, end) pairs SE $= \{(s_1, e_1), ..., (s_n, e_n)\}$ which contains the pairs of the function start and the function ends of the corresponding functions in $\{f_1, ..., f_n\}$.

Function scope identification: This is the hardest problem in the function identification task. In this problem, we need to find out the set $\{(f_{1,s_1}, ..., f_{1,e_1}), ..., (f_{n,s_n}, ..., f_{n,e_n})\}$ which specifies the machine instructions in each function $f_1, ..., f_n$ in the given binary program P. It is apparent that the solution of this problem covers those of three aforementioned problems. We note that our proposed CPN addresses this hardest problem, hence inherently offering solutions for the other problems.

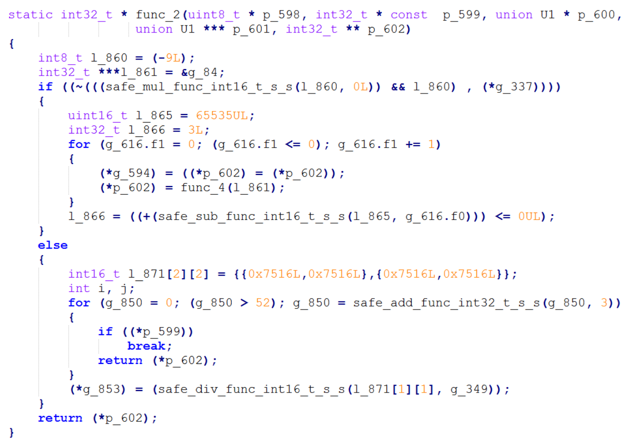
B. Running Example

In Fig. 1, we show a typical example of a function generated by Csmith. According to our observation, source codes generated by Csmith have a wide range of variety in both control and data flows. Fig. 2 shows the assembly code of the source code in Fig. 1 which was compiled with the optimization levels O1 and Ox on the Windows platform. It can be observed that the entry pattern for each optimization level is different. Besides that the assembly code corresponding with the option Ox has three rets (i.e., return instruction) and the last ret is the real end point while the assembly code corresponding with the option O1 has only one ret. We further observe that in the real generated binary codes, the patterns for the entry point vary in a wide range and can start with *push*, *mov*, *movsx*, *inc*, *cmp*, *or*, *and*, etc. These make the task of function identification very challenging.

C. Challenges of The Function Identification Task

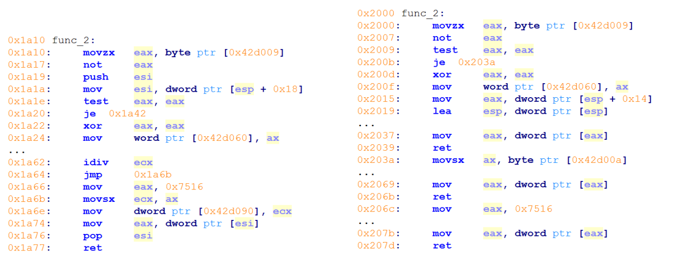
In what follows, we list some challenges of the task of function identification. These challenges originate from various behaviors of compilers when compiling source codes under various combinations of optimization levels (e.g., O1, O2, and O3 or Ox), processor architectures (e.g., x86 and x64), and platforms (e.g., Windows or Linux).

Not every machine instruction belongs to a function: Compilers may introduce additional instructions for alignment and padding between or within a function which leads to some machine instructions that do not belong to any function.



```c
static int32_t * func_2(uint8_t * p_598, int32_t * const  p_599, union U1 * p_600,
                        union U1 *** p_601, int32_t ** p_602)
{
    int8_t l_860 = (-9L);
    int32_t ***l_861 = &g_84;
    if ((~(((safe_mul_func_int16_t_s_s(l_860, 0L)) && l_860) , (*g_337))))
    {
        uint16_t l_865 = 65535UL;
        int32_t l_866 = 3L;
        for (g_616.f1 = 0; (g_616.f1 <= 0); g_616.f1 += 1)
        {
            (*g_594) = ((*p_602) = (*p_602));
            (*p_602) = func_4(l_861);
        }
        l_866 = ((+(safe_sub_func_int16_t_s_s(l_865, g_616.f0))) <= 0UL);
    }
    else
    {
        int16_t l_871[2][2] = {{0x7516L,0x7516L},{0x7516L,0x7516L}};
        int i, j;
        for (g_850 = 0; (g_850 > 52); g_850 = safe_add_func_int32_t_s_s(g_850, 3))
        {
            if ((*p_599))
                break;
            return (*p_602);
        }
        (*g_853) = (safe_div_func_int16_t_s_s(l_871[1][1], g_349));
    }
    return (*p_602);
}
```

Figure 1. An example source code of a function generated by Csmith.



```
0x1a10 func_2:
0x1a10:     movzx   eax, byte ptr [0x42d009]
0x1a17:     not     eax
0x1a19:     push    esi
0x1a1a:     mov     esi, dword ptr [esp + 0x18]
0x1a1e:     test    eax, eax
0x1a20:     je  0x1a42
0x1a22:     xor     eax, eax
0x1a24:     mov     word ptr [0x42d060], ax
...
0x1a62:     idiv    ecx
0x1a64:     jmp     0x1a6b
0x1a66:     mov     eax, 0x7516
0x1a6b:     movsx   ecx, ax
0x1a6e:     mov     dword ptr [0x42d090], ecx
0x1a74:     mov     eax, dword ptr [esi]
0x1a76:     pop     esi
0x1a77:     ret
```

(3.a) Compiled using MVS with O1.

```
0x2000 func_2:
0x2000:     movzx   eax, byte ptr [0x42d009]
0x2007:     not     eax
0x2009:     test    eax, eax
0x200b:     je  0x203a
0x200d:     xor     eax, eax
0x200f:     mov     word ptr [0x42d060], ax
0x2015:     mov     eax, dword ptr [esp + 0x14]
0x2019:     lea     esp, dword ptr [esp]
...
0x2037:     mov     eax, dword ptr [eax]
0x2039:     ret
0x203a:     movsx   ax, byte ptr [0x42d00a]
...
0x2069:     mov     eax, dword ptr [eax]
0x206b:     ret
0x206c:     mov     eax, 0x7516
...
0x207b:     mov     eax, dword ptr [eax]
0x207d:     ret
```

(3.b) Compiled using MVS with Ox.

Figure 2. The assembly codes of the example source code in Fig. 1 compiled using Microsoft Visual Studio (MVS) with the x86 architecture and Window platform under the optimization levels O1 (left) and Ox (right).

Functions may be non-contiguous: There may exist gaps between functions which can jump tables, data, or even instructions for completely different functions. In addition, as observed by [23], function sharing code can also lead to non-contiguous functions.

Functions may have multiple entries: High-level languages use functions as an abstraction with a single entry. When compiled, however, functions may have multiple entries as a result of specialization. In addition, the number of patterns for function start can be enormous and varied according to optimization levels, processor architectures, and platforms.

IV. Code Pointer Network for The Function Identification Problem

In this section, we present our proposed Code Pointer Network (CPN) that can tackle the function identification task. We start with the discussion of how to process instructions to input to our CPN, followed by technical details for the training and testing procedures.

A. Processing input statement

We compiled the source code programs for the x86 architecture with three different optimization levels (O1, O2 and Ox). Each machine instruction may have a different size which can be 4, 5, 6, 7, 8 bytes or even more. To the best of our knowledge, the first 4 bytes in x86 architecture mostly contains the crucial information for a machine instruction (i.e., the opcode and other crucial addresses), hence before feeding these machine instructions to the CPN, we preprocess them as follows: (1) keep the first 4 bytes from the left for machine instructions which are longer than 4 bytes and (2) padding 0 to the right of the machine instructions which have fewer than 4 bytes. For example, the machine instruction "*mov dword ptr [0x42b008], 0x0*" has the corresponding value "*C70508B0420000000000*" in hex format which is 10 bytes long, we then remove the last 6 bytes and keep the first 4 bytes to get the numerical value "*C70508B0*". For the machine instruction "*xor al, al*" which contains 2 bytes "*32C0*" in the hex format, we then pad with 0 to fill in the third and fourth bytes and gain the numerical value "*32C00000*".

B. Code Pointer Network

1) Training procedure: The Code Pointer Network architecture is depicted in Fig. 3. Our CPN takes as input a binary program including a sequence of many machine instructions which may belong to different functions. The task of the CPN is to read out the scope of the first function in the input binary program. To this end, as in a typical pointer network, our proposed CPN consists of two components: one encoder and one decoder. The encoder's task is to encode the sequence of machine instructions in a binary program, while the decoder tries to decode the encoded output of the encoder to read out the indexes of the machine instructions in the first function in the given binary program. In addition, to signal the end of the function, we introduce the specific symbol *EOF* whose value is $\mathbf{i}_E$ which is randomly initialized. This value $\mathbf{i}_E$ is inputted to the CPN encoder right after the last instruction in the binary program (cf. Fig. 3).

To reduce the sequence length of the CPN encoder, at a time, we input two consecutive functions including gaps between functions if existing in a binary program to the encoder. For example, as shown in Fig. 3, a binary program that consists of 4 functions forms 3 pairs of two consecutive functions (i.e., (gap ,func1, gap, func2, gap), (gap, func2, gap, func3, gap), and (gap, func3, gap, func4, gap)) and the encoder takes each pair at a time. Let us now define the input sequence $\mathrm{I} = \{\mathbf{i}_1, \mathbf{i}_2, ..., \mathbf{i}_{l+1}\}$ (i.e., the machine instructions in a pair) and the output sequence $\mathbf{O} = \{n_1, n_2, ..., n_m, n_{m+1}\}$ where n_{m+1} specifies the index of the symbol EOF and which includes the indexes of machine instructions in the first function in the input pair. We learn the model parameters θ by maximizing the following conditional probabilities over the training set $\mathcal{D}$ which includes the pair I and the indexes of machine instructions in its first function O:

$$\theta^* = \underset{\theta}{\operatorname{argmax}} \sum_{(\mathrm{I},\mathbf{O}) \in \mathcal{D}} \log p(\mathbf{O}|\mathrm{I};\theta)$$

where we have defined

$$p(\mathbf{O}|\mathrm{I};\theta) = \prod_{k=1}^{m} p(n_{k+1}|n_1, ..., n_k, \mathrm{I};\theta) \qquad (1)$$

We now denote $\{\mathbf{h}_1, \mathbf{h}_2, ..., \mathbf{h}_{l+1}\}$ and $\{\mathbf{s}_0, \mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_m\}$ with $\mathbf{s}_0 = \mathbf{s}$ as the encoder hidden states and decoder hidden states, respectively. Since $\mathbf{s}_k$ is a function of $n_1, ..., n_k, \mathrm{I}$ or a

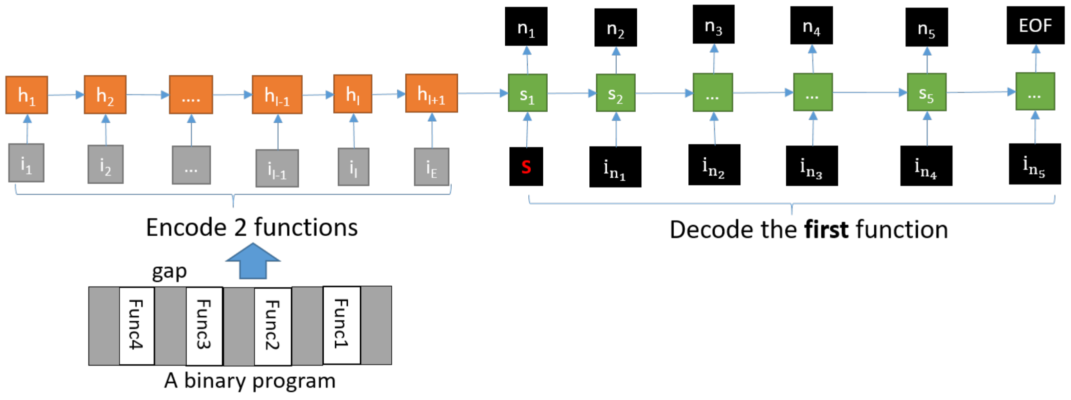


Figure 3. The Code Pointer Network (CPN) architecture.

lossy summary of this sequence, we can reasonably simplify $p(\mathbf{O}|\mathrm{I};\theta)$ as follows:

$$p(\mathbf{O}|\mathrm{I};\theta) = \prod_{k=1}^{m} p(n_{k+1}|\mathbf{s}_k;\theta)$$

$$\log p(\mathbf{O}|\mathrm{I};\theta) = \sum_{k=1}^{m} \log p(n_{k+1}|\mathbf{s}_k;\theta)$$

To define the probability $\log p(n_{k+1}|\mathbf{s}_k;\theta)$ where $n_k \in \{1,2,\dots,l+1\}$, we first compute the alignment scores between the decoder hidden state $\mathbf{s}_k$ and the encoder hidden states $\mathbf{h}_j$, $\forall j = 1,\dots,l+1$:

$$\mathbf{a}_{jk} = \mathbf{v}^{\mathrm{T}}\tanh(\mathrm{U}_h\mathbf{h}_j + \mathrm{U}_s\mathbf{s}_k) \quad (2)$$

$$\text{or } \mathbf{a}_{jk} = \mathbf{v}^{\mathrm{T}}\tanh\left(\mathrm{U}\begin{bmatrix}\mathbf{h}_j\\ \mathbf{s}_k\end{bmatrix}\right) \quad (3)$$

where the vector $\mathbf{v}$ and the matrices U, U_h, Us are learnable parameters.

We then apply the softmax to $[\mathbf{a}_{jk}]_{j=1}^{l+1}$ to gain the vector $\mathbf{b}_k$ and define $p(n_{k+1}|\mathbf{s}_k;\theta)$ as the n_{k+1}-th element in this vector:

$$\mathbf{b}_k = \text{softmax}\left([\mathbf{a}_{jk}]_{j=1}^{l+1}\right)$$

$$p(n_{k+1}|\mathbf{s}_k;\theta) = \mathbf{b}_{k,n_{k+1}}$$

2) Predicting procedure: In the predicting procedure, given a specific binary program, we first input this binary program into the encoder of the trained model to read out the first function (machine instructions and their positions). The detected function is then eliminated from the binary program and the remaining binary code is once again inputted to the CPN. This process is repeated until the last function. We visualize the process for the predicting procedure in Fig. 5.

V. Experiments

In this section, we present the experimental results of our proposed Code Pointer Network compared with the Bidirectional recurrent neural network (the Bidirectional RNN). We also investigate the performance of our CPN with various RNN cells (e.g., Long Short Term Memory (LSTM) and Gated Recurrent Unit (GRU) cells) and alignment score formulas as shown in Eqs. (2 and 3). We note that we could not experiment with ByteWeight [10] since this method is not scalable enough for our data (i.e, ByteWeight took about 587 compute-hours for training a dataset of 2,064 binaries [13] while our data has 120,000 random C/C++ programs). In addition, the Bidirectional RNN [13] has been proven to outperform ByteWeight.

A. Data Collection

We used Csmith [16], which is a well-known tool for generating random C/C++ programs conforming dynamically and statically to the C99 standard, to generate 120,000 random C/C++ programs. Each generated program has a number of functions in the range {2, 3, 4, 5}. Binaries were then compiled from the source programs using Microsoft Visual Studio in the debug mode with one of three different optimization levels O1 (for creating the smallest code), O2 (for creating the fastest code), and Ox (with full optimization options including smallest and fastest code) under x86 (32 bit) architecture. Finally, we used DIA2Dump[1] to read the debug files (i.e., *.pdb) for creating the labelled dataset.

B. Experimental Setting

We divided the binaries into three random parts; the first part contains 80% of the binaries used for training, the second part contains 10% of the binaries used for testing, and the third part contains 10% of the binaries for validation. For each the method, we trained the competitive methods over 40,000 iterations.

For the Bidirectional RNN, we used identical settings and the architecture proposed in [13]. In particular, we chopped the binaries into chunks of 1,000 bytes. This means that we run recurrent neural networks forward and backward on a 1000-byte sequence from the corresponding binaries. The size of the hidden state for the forward and backward RNN is 32. Then the forward and backward RNN are concatenated to feed into a linear transformation and the soft-max function. This process produces a probability distribution to identify whether a byte corresponds to the beginning (or end) of a

[1]https://docs.microsoft.com/en-us/visualstudio/debugger/debug-interface-access/debug-interface-access-sdk

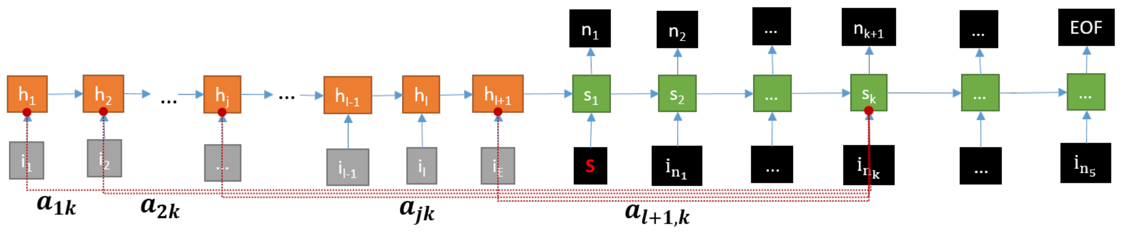


Figure 4. The alignment scores in the CPN network.

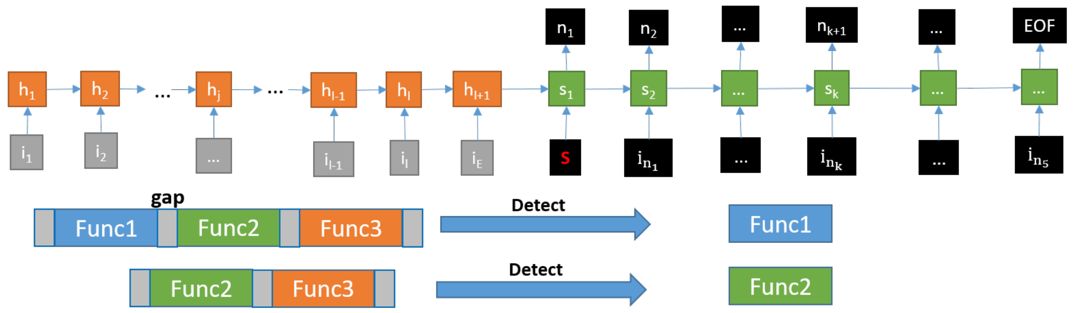


Figure 5. The predicting procedure of the CPN network.

function or not. We employed the RMSprop optimizer with the learning rate varying in the range of {0.01, 0.1} and the batch size is set to 32. We trained the models for function start identification task and function end identification task separately. The Bidirectional RNN proposed in [13] is not directly applicable to function scope identification. To make it applicable to this task, we first paired the corresponding function starts and function ends detected by two Bidirectional RNN and assume that each pair forms a boundary for a function where all machine instructions in this boundary are counted as that in this function and counted it as a correct function prediction if this matches exactly a real function.

For our model in the training process, we used the encoder with a sequence of 100 hidden states where the hidden state size is 128. For each binary, we concatenate sequentially two functions as input for the Code Pointer Network in the encoder process. We employed the Adam optimizer with the default learning rate 0.001 and the batch size 128. In addition, we applied Max-norm regularization to prevent the over-fitting problem when training the model. Because our CPN solves the function scope identification task, it can be inherently applicable to the function start and function end identification tasks.

We implemented the Code Pointer Network and the Bidirectional RNN in Python using Tensorflow [24], an open-source software library for Machine Intelligence developed by the Google Brain Team. We ran our experiments on an Intel Xeon Processor E5-1660 which has 8 cores at 3.0 GHz and 128 GB of RAM.

C. Metrics

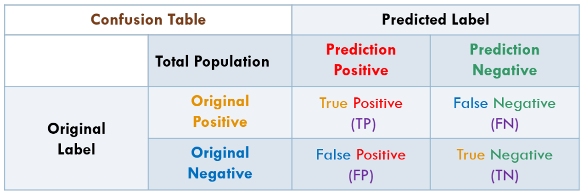
In order to evaluate performances of function identification methods, we employ three measures including recall (R), precision (P) and F1-score (F1) which are widely used to report predictive performances on imbalanced datasets. This is due to the fact that the number of function starts, function ends, etc., are fewer than the number of machine instructions. Given a dataset with two kinds of labels: positive and negative labels, the precision is the fraction of the number of true positive instances among the number of original positive instances. The recall is the faction of the number of true positive instances among the number of predicted positive instances. The F1-score is the most important measure which aggregates both precision and recall. The recall, precision and F1-score have the following forms:

$$P = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}$$

$$R = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

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

where TP, FP and FN are the number of true positives, false positives and false negatives, respectively which can be defined using a confusion table as shown in Fig. 6.



Confusion Table		Predicted Label	
	Total Population	Prediction Positive	Prediction Negative
Original Label	Original Positive	True Positive (TP)	False Negative (FN)
	Original Negative	False Positive (FP)	True Negative (TN)

Figure 6. The confusion table.

Table I
COMPARISON OF OUR CODE POINTER NETWORK AND THE BIDIRECTIONAL RNN. THE BEST RESULTS (%) ARE EMPHASIZED IN **BOLD**.

Optimization	**O1**								
Methods	**Function Start**			**Function End**			**Function Scope**		
	R	**P**	**F1**	**R**	**P**	**F1**	**R**	**P**	**F1**
CPN	**96.53%**	97.26%	**96.89%**	94.09%	**94.64%**	**94.36%**	**92.77%**	**93.15%**	**92.96%**
Bidirectional RNN	86.30%	**98.14%**	91.84%	**96.42%**	81.58%	88.38%	81.56%	82.17%	81.87%
Optimization	**O2**								
Methods	**Function Start**			**Function End**			**Function Scope**		
	R	**P**	**F1**	**R**	**P**	**F1**	**R**	**P**	**F1**
CPN	**95.18%**	93.27%	**94.23%**	**89.60%**	**87.87%**	**88.73%**	**87.55%**	**88.20%**	**87.87%**
Bidirectional RNN	82.48%	**98.16%**	89.63%	87.86%	72.52%	79.45%	73.14%	78.43%	75.69%
Optimization	**Ox**								
Methods	**Function Start**			**Function End**			**Function Scope**		
	R	**P**	**F1**	**R**	**P**	**F1**	**R**	**P**	**F1**
CPN	**94.43%**	**93.77%**	**94.10%**	**89.07%**	**88.57%**	**88.82%**	**87.34%**	**88.01%**	**87.67%**
Bidirectional RNN	74.39%	80.02%	77.10%	79.62%	70.14%	74.58%	71.18%	73.21%	72.18%

Table II
COMPARISON OF THE VARIANTS OF CPN USING DIFFERENT TYPES OF RNN CELLS INCLUDING LSTM, GRU AND THE STANDARD RNN CELL. THE BEST RESULTS (%) ARE EMPHASIZED IN **BOLD**.

Optimization	**O1**								
Methods	**Function Start**			**Function End**			**Function Scope**		
	R	**P**	**F1**	**R**	**P**	**F1**	**R**	**P**	**F1**
CPN-LSTM-01	**95.32%**	96.42%	**95.87%**	**92.75%**	**90.85%**	**91.79%**	**90.27%**	**91.19%**	**90.73%**
CPN-GRU-01	92.78%	**96.45%**	94.58%	90.14%	90.52%	90.33%	84.91%	87.40%	86.14%
CPN-RNN-01	91.32%	96.16%	93.68%	90.07%	90.42%	90.24%	85.26%	86.54%	85.90%
Optimization	**O2**								
Methods	**Function Start**			**Functions End**			**Function Scope**		
	R	**P**	**F1**	**R**	**P**	**F1**	**R**	**P**	**F1**
CPN-LSTM-01	**93.09%**	**93.38%**	**93.23%**	87.04%	87.36%	87.20%	**86.75%**	**87.96%**	**87.35%**
CPN-GRU-01	93.03%	93.25%	93.14%	**87.48%**	**87.69%**	**87.58%**	84.35%	85.45%	84.90%
CPN-RNN-01	91.76%	92.86%	92.30%	86.52%	87.63%	87.07%	83.41%	84.24%	83.82%
Optimization	**Ox**								
Methods	**Function Start**			**Functions End**			**Function Scope**		
	R	**P**	**F1**	**R**	**P**	**F1**	**R**	**P**	**F1**
CPN-LSTM-01	**92.47%**	**93.16%**	**92.81%**	**86.35%**	**85.54%**	**86.94%**	**86.56%**	**87.71%**	**87.12%**
CPN-GRU-01	90.81%	93.94%	92.35%	85.87%	88.71%	87.27%	84.59%	84.73%	84.65%
CPN-RNN-01	90.53%	92.43%	91.47%	84.95%	87.56%	86.24%	83.32%	84.17%	83.74%

Table III
COMPARISON OF VARIANTS OF CPN USING EQ. (2) (CPN-LSTM-01) AND EQ. (3) (CPN-LSTM-02) FOR COMPUTING ALIGNMENT SCORE. THE BEST RESULTS (%) ARE EMPHASIZED IN **BOLD**.

Optimization	**O1**								
Methods	**Function Start**			**Function End**			**Function Scope**		
	R	**P**	**F1**	**R**	**P**	**F1**	**R**	**P**	**F1**
CPN-LSTM-01	95.32%	96.42%	95.87%	92.75%	90.85%	91.79%	90.27%	91.19%	90.73%
CPN-LSTM-02	**96.53%**	**97.26%**	**96.89%**	**94.09%**	**94.64%**	**94.36%**	**92.77%**	**93.15%**	**92.96%**
Optimization	**O2**								
Methods	**Function Start**			**Function End**			**Function Scope**		
	R	**P**	**F1**	**R**	**P**	**F1**	**R**	**P**	**F1**
CPN-LSTM-01	93.09%	**93.38%**	93.23%	87.04%	87.36%	87.20%	86.75%	87.96%	87.35%
CPN-LSTM-02	**95.18%**	93.27%	**94.23%**	**89.60%**	**87.87%**	**88.73%**	**87.55%**	**88.20%**	**87.87%**
Optimization	**Ox**								
Methods	**Function Start**			**Function End**			**Function Scope**		
	R	**P**	**F1**	**R**	**P**	**F1**	**R**	**P**	**F1**
CPN-LSTM-01	92.47%	93.16%	92.81%	86.35%	85.54%	86.94%	86.56%	87.71%	87.12%
CPN-LSTM-02	**94.43%**	**93.77%**	**94.10%**	**89.07%**	**88.57%**	**88.82%**	**87.34%**	**88.01%**	**87.67%**

D. Experimental Results

Code Pointer Network versus Bidirectional RNN: We compare our method using Eq. (3) for computing alignment scores and Long Short Term Memory (LSTM) for RNN cell with the Bidirectional RNN proposed in [13]. The experimental results show that our proposed method achieves better performance in most cases in terms of predictive performance and training time. In particular, Table I indicates that our proposed CPN achieved better predictive performance (i.e., R : Recall, P: Precision, and F1: F1-score) with a wide margin in most cases, especially for the highest optimization level Ox our CPN significantly outperformed the baseline in all measures. Regarding the training time, our CPN is approximately three times faster than the baseline. In particular, with the same number of iterations (i.e., 40,000 iterations), the CPN took around 2 hours to finish, while the baseline took around 6 hours.

Variations in RNN cells: In Table II, we compare the performances of our CPN using different RNN cells such as Long Short Term Memory (CPN-LSTM-01) and Gated Recurrent Unit (CPN-GRU-01) with the basic RNN cell (CPN-RNN-01) with Eq. (2) for computing alignment score. It can be observed that LSTM achieved better performance than GRU which in turn performed better than the basic RNN cell in most cases.

Variation in attention mechanism techniques: In Table III, we compare the performances of our CPN using different formulas for computing alignment score as in Eq. (2) (CPN-LSTM-01) with Eq. (3) (CPN-LSTM-02) while employing the LSTM cell. The experimental results show that CPN-LSTM-02 obtained better performances in most cases compared with CPN-LSTM-01.

VI. Conclusion

In this paper, we have proposed the novel Code Pointer Network for dealing with the function identification problem, a preliminary and significant step in binary analysis for many security applications such as malware detection, common vulnerability detection and binary instrumentation. Specifically, the Code Pointer Network leverages the underlying idea of a pointer network in order to tackle the function scope identification, the hardest and most crucial task in function identification. The experimental results show that the Code Pointer Network can achieve the state-of-the-art performances in terms of efficiency and efficacy.

References

[1] J. Caballero, N. M. Johnson, S. McCamant, and D. Song, "Binary code extraction and interface identification for security applications," in *Proceedings of the 17th Network and Distributed System Security Symposium*, 2010.

[2] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically patching errors in deployed software," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[3] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, 2015.

[4] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "Pebil: Efficient static binary instrumentation for linux," *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2010.

[5] A. Prakash, X. Hu, and H. Yin, "vfguard: Strict protection for virtual function calls in cots c++ binaries," in *National Diabetes Services Scheme (NDSS)*, 2015.

[6] T. Le, T. Nguyen, T. Le, P. Montague, O. De Vel, L. Qu, and D. Phung, "Maximal divergence sequential autoencoder for binary software vulnerability detection," in *In International Conference on Learning Representations*, 2019.

[7] T. Nguyen, T. Le, K. Nguyen, O. De Vel, P. Montague, J. Grundy, and D. Phung, "Deep cost-sensitive kernel machine for binary software vulnerability detection," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2020.

[8] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, 2004.

[9] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *Proceedings of the 22Nd USENIX Conference on Security*, 2013.

[10] T. Bao, J. Burket, and M. Woo, "Byteweight: Learning to recognize functions in binary code," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[11] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, 2011.

[12] D. Brumley, I. Jager, T. Avgerinos, and dward J. Schwartz, "Bap: A binary analysis platform," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, 2011.

[13] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[14] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[15] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds. Curran Associates, Inc., 2015, pp. 2692–2700.

[16] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," *SIGPLAN Not.*, vol. 46, no. 6, pp. 283–294, Jun. 2011.

[17] O. Vinyals, S. Bengio, and M. Kudlur, "Order matters: Sequence to sequence for sets," *arXiv preprint arXiv:1511.06391*, 2015.

[18] C. Gulcehre, S. Ahn, R. Nallapati, B. Zhou, and Y. Bengio, "Pointing the unknown words," *arXiv preprint arXiv:1603.08148*, 2016.

[19] V. Nguyen, T. Le, T. Le, K. Nguyen, O. DeVel, P. Montague, L. Qu, and D. Phung, "Deep domain adaptation for vulnerable code function identification," in *2019 International Joint Conference on Neural Networks (IJCNN)*, 2019, pp. 1–8.

[20] V. Nguyen, T. Le, O. De Vel, P. Montague, J. Grundy, and D. Phung, "Dual-component deep domain adaptation: A new approach for cross project software vulnerability detection." in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2020.

[21] N. E. Rosenblum, X. Zhu, B. P. Miller, and K. Hunt, "Learning to analyze binary computer code." in *AAAI*, 2008, pp. 798–804.

[22] V. Nguyen, T. Le, T. Le, K. Nguyen, O. De Vel, P. Montague, J. Grundy, and D. Phung, "Code action network for binary function scope identification," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2020.

[23] L. C. Harris and B. P. Miller, "Practical analysis of stripped binary code," *SIGARCH Comput. Archit. News*, vol. 33, no. 5, pp. 63–68, Dec. 2005.

[24] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.