Code Action Network for Binary Function Scope Identification

Van Nguyen[1(✉)], Trung Le[1(✉)], Tue Le[2(✉)], Khanh Nguyen[2(✉)], Olivier de Vel[3(✉)], Paul Montague[3(✉)], John Grundy[1(✉)], and Dinh Phung[1(✉)]

[1] Monash University, Clayton, Australia
{van.nk,trunglm,john.grundy,dinh.phung}@monash.edu
[2] AI Research Lab, Trusting Social, Melbourne, Australia
tue.le.ict@jvn.edu.vn, khanh@trustingsocial.com
[3] Defence Science and Technology Group, Canberra, Australia
{Olivier.DeVel,Paul.Montague}@dst.defence.gov.au

Abstract. Function identification is a preliminary step in binary analysis for many applications from malware detection, common vulnerability detection and binary instrumentation to name a few. In this paper, we propose the Code Action Network (CAN) whose key idea is to encode the task of function scope identification to a sequence of three action states NI (i.e., next inclusion), NE (i.e., next exclusion), and FE (i.e., function end) to efficiently and effectively tackle function scope identification, the hardest and most crucial task in function identification. A bidirectional Recurrent Neural Network is trained to match binary programs with their sequence of action states. To work out function scopes in a binary, this binary is first fed to a trained CAN to output its sequence of action states which can be further decoded to know the function scopes in the binary. We undertake extensive experiments to compare our proposed method with other state-of-the-art baselines. Experimental results demonstrate that our proposed method outperforms the state-of-the-art baselines in terms of predictive performance on real-world datasets which include binaries from well-known libraries.

Keywords: Cyber security · Function scope identification · Machine learning · Deep learning

1 Introduction

In computer security, we often encounter situations 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 applications such as malware detection, common vulnerability detection [9], and etc. Function identification is usually the first step in many binary analysis methods. This aims to specify function scopes in a binary and is a building block to a diverse range of application domains including binary instrumentation [5], vulnerability research [10] and binary protection structures with Control-Flow Integrity. In both binary

H. W. Lauw et al. (Eds.): PAKDD 2020, LNAI 12084, pp. 712–725, 2020.
https://doi.org/10.1007/978-3-030-47426-3_55

analysis and function identification, tackling the loss of high-level semantic structures in binaries which results from compilers during the process of compilation is likely the most challenging problem.

There have been many effective methods for dealing with the function identification problem from heuristic solutions (statistical methods for binary analysis) to complicated approaches employing machine learning or deep learning techniques. In an early work, Kruegel et al. [4] through his research which leveraged statistical methods with control flow graphs concluded that the task of function start identification can be trivially solved for regular binaries. However, later research in [14] argued that this task is non-trivial and complex in some specific cases wherein it is too challenging for heuristics-based methods to discover all function boundaries. Other influential works and tools that rely on signature database and structural graphs include IDA Pro, Dyninst, (Binary Analysis Platform) BAP, and Nucleus [1]. Andriesse et al. [1] has recently proposed a new signature-less approach to function detection for stripped binaries named Nucleus which is based on structural Control Flow Graph analysis. More specifically, Nucleus identifies functions in the intraprocedural control flow graph (ICFG) by analyzing the control flow between basic blocks, based on the observation that intraprocedural control flow tends to use different types and patterns of control flow instructions than inter-procedural control flow.

Machine learning has been applied to binary analysis and function identification in particular. The seminal work of [11] modeled function start identification as a Conditional Random Field (CRF) in which binary offsets and a number of selected patterns appear in the CRF. Since the inference on a CRF is very expensive, though feature selection and approximate inference were adopted to speed up this model, its computational complexity is still very high. ByteWeight [2] is another successful machine learning based method for function identification aiming to learn 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 and Binary Analysis Platform (BAP), it is not scalable enough for even medium-sized datasets [12].

Deep learning has undergone a renaissance in the past few years, achieving breakthrough results in multiple application domains such as visual object recognition [3], language modeling [13], and software vulnerability detection [6–8]. The study in [12] is the first work which 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 [2] while requiring much less training time. However, to address the boundary identification problem with [12], a simple heuristic to pair adjacent function starts and function ends was used (see Section 5.3 in that paper). Consequently, this approach is not able to efficiently utilize the context information of consecutive bytes and machine instructions in a function and the pairing procedure might lead to inconsistency since the networks for function start and end were trained independently. Furthermore, this method cannot address the function scope identification problem, the hardest and most essential sub problem in function identification, wherein the scope (i.e., the addresses of all machine instructions in a function) of each function must be specified.

Inspired from the idea of a Turing machine, we imagine a memory tape consisting of many cells on which machine instructions of a binary are stored. The head is first pointed to the first machine instruction located in the first cell. Each machine instruction is assigned to an action state in the action state set {NI, NE, FE} depending on its nature. After reading the current machine instruction and assigning the corresponding action state to it, the head is moved to the next cell and this procedure is halted as we reach the last cell in the tape (see Sect. 3.1). Eventually, the sequence of machine instructions in a given binary is translated to the corresponding sequence of action states. Based on this incentive, in this paper, we propose a novel method named the *Code Action Network* (CAN) whose underlying idea is to equivalently transform the task of function scope identification to learning a sequence of action states. A bidirectional Recurrent Neural Network is trained to match binary programs with their corresponding sequences of action states. To predict function scopes in any binary, the binary is first fed to a trained CAN to output its corresponding sequence of action states on which we can then work out function scopes in the binary. The proposed CAN can tackle binaries for which there exist external gaps between functions and internal gaps inside functions wherein each internal gap in a function does not contain instructions from other functions. By default, our CAN named as CAN-B operates at the byte level and can cope with all binaries that satisfy the aforementioned condition. However, for the binaries that can be further disassembled into machine instructions, another variant named as CAN-M is able to operate at the machine instruction level. CAN-M can efficiently exploit the semantic relationship among bytes in an instruction and instructions in a function as well as requiring much shorter sequence length compared with the Bidirectional RNN in [12] which also works at the byte level. In addition, our proposed CAN-B and CAN-M can directly address the function scope identification task, hence inherently offering the solution for other simpler tasks including the function start/end/boundary identifications.

We undertake extensive experiments to compare our proposed CAN-B and CAN-M with state-of-the-art methods including IDA, the Bidirectional RNN, ByteWeight no-RFCR and ByteWeight on the dataset used in [2,12]. The experimental results show that our proposed CAN-B and CAN-M outperform the baselines on function start, function end and function boundary identification tasks as well as achieving very good performance on function scope identification and also surpass the Nucleus [1] on this task. Our proposed methods slightly outperform the Bidirectional RNN proposed in [12] on the function start and end identification tasks, but significantly surpass this method on the function boundary identification task – the more important task. This demonstrates the capacity of our methods in efficiently utilizing the contextual relationship carried in consecutive machine instructions or bytes to properly match the function start and end entries for this task. As expected, our CAN-M obtains the best predictive performances on most experiments and is much faster than the Bidirectional RNN proposed in [12]. Particularly, CAN-M takes about 1 hour for training with 20,000 iterations which is nearly 4 times faster than the Bidirectional RNN proposed in [12] using the same number of iterations for training and the same number of bytes for handling input. This is due to the fact that CAN-M operates at the machine instruction level, while the Bidirectional RNN proposed in [12] operates at the byte level.

We also do error analysis to qualitatively compare our CAN-M and CAN-B with the baselines. We observe that there are a variety of instruction styles for the function start and function end (e.g., in the experimental dataset, there are a thousand different function start styles and function end styles). In their error analyses, Shin et al. [12] and Bao et al. [2] mentioned that for functions which encompass several function start styles or function end styles, their proposed methods tend to make mistakes in predicting the function start or end bytes with many false positives and negatives. However, it is not the case for our proposed methods, since we further observe that for the functions which contain more than one function start style or function end style which account for 98.38% and 28% of the testing set respectively, our proposed CAN-M has 0.24% and 1.09% false positive rates respectively.

2 The Function Identification Problem

This section discusses the function identification problem. We begin with definitions of the sub problems in the function identification problem, followed by an example of source code in the C language and its binaries compiled with optimization levels O1 using *gcc* on the Linux platform for the x86-64 architecture.

2.1 Problem Definitions

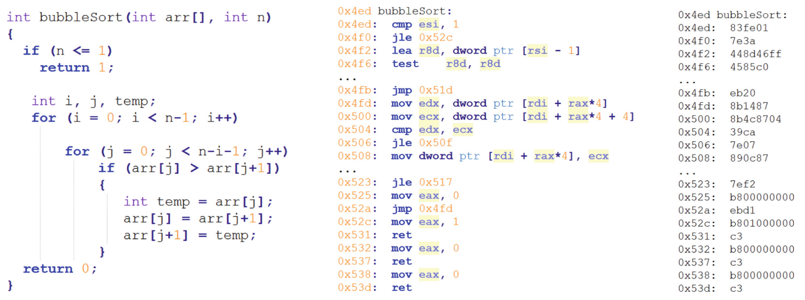
Given a binary program P, our task is to identify the necessary information (e.g., function starts, function ends) in its n functions $\{f_1, ..., f_n\}$ which is initially unknown. Depending on the nature of information we need from $\{f_1, ..., f_n\}$, we can categorize the task of function identification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

Function Start/End/boundary Identification. In the first problem, we need to specify the set $\mathrm{S} = \{s_1, ..., s_n\}$ which contains the start instruction byte for each of the corresponding functions in $\{f_1, ..., f_n\}$. If a function (e.g. f_i) has multiple start points, s_i will be the first start instruction byte for f_i. In the second problem, we need to identify the set $\mathrm{E} = \{e_1, ..., e_n\}$ which contains the end instruction byte for each of the corresponding functions in $\{f_1, ..., f_n\}$. If a function (e.g. f_i) has multiple exit points, e_i will be the last end instruction byte for f_i. In the last problem, we have to point out the set of (start, end) pairs $\mathrm{SE} = \{(s_1, e_1), ..., (s_n, e_n)\}$ which contains the pairs of the function start and the function end for each 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 instruction bytes in each function $f_1, ..., f_n$ in the given binary program P. Here we note that because functions may be not contiguous, the instruction bytes $(f_{i,s_i}, ..., f_{i,e_i})$ may also be not contiguous. It is apparent that the solution of this problem covers the three aforementioned problems. Since our proposed CAN addresses this problem, it inherently offers solutions for the other problems.

2.2 Running Example

In Fig. 1, we show an example of a short source code fragment for a function in the C programming language, the corresponding assembly code in the machine instruction and corresponding hexadecimal mode of the binary code respectively, which was compiled using *gcc* with the optimization level O1 for the x86-64 architecture on the Linux platform. We further observe that in real binary code, the patterns for the entry point vary over a wide range and can start with *push*, *mov*, *movsx*, *inc*, *cmp*, *or*, *and*, etc. In the example, the assembly code corresponding with the optimization level O1 on Linux has three *ret* statements. Furthermore, in real binary code, the ending point of a function can vary in pattern beside the *ret* pattern. These make the task of function identification very challenging. For the challenges of the function scope identification task, we refer the readers to [2,12] and the discussions therein.



```c
int bubbleSort(int arr[], int n)
{
  if (n <= 1)
    return 1;

   int i, j, temp;
   for (i = 0; i < n-1; i++)

       for (j = 0; j < n-i-1; j++)
           if (arr[j] > arr[j+1])
           {
              int temp = arr[j];
              arr[j] = arr[j+1];
              arr[j+1] = temp;
           }
  return 0;
}
```

```
0x4ed bubbleSort:
0x4ed:  cmp esi, 1
0x4f0:  jle 0x52c
0x4f2:  lea r8d, dword ptr [rsi - 1]
0x4f6:  test    r8d, r8d
...
0x4fb:  jmp 0x51d
0x4fd:  mov edx, dword ptr [rdi + rax*4]
0x500:  mov ecx, dword ptr [rdi + rax*4 + 4]
0x504:  cmp edx, ecx
0x506:  jle 0x50f
0x508:  mov dword ptr [rdi + rax*4], ecx
...
0x523:  jle 0x517
0x525:  mov eax, 0
0x52a:  jmp 0x4fd
0x52c:  mov eax, 1
0x531:  ret
0x532:  mov eax, 0
0x537:  ret
0x538:  mov eax, 0
0x53d:  ret
```

```
0x4ed bubbleSort:
0x4ed:  83fe01
0x4f0:  7e3a
0x4f2:  448d46ff
0x4f6:  4585c0
...
0x4fb:  eb20
0x4fd:  8b1487
0x500:  8b4c8704
0x504:  39ca
0x506:  7e07
0x508:  890c87
...
0x523:  7ef2
0x525:  b800000000
0x52a:  ebd1
0x52c:  b801000000
0x531:  c3
0x532:  b800000000
0x537:  c3
0x538:  b800000000
0x53d:  c3
```

Fig. 1. Example source code of a function in the C language programming (Left), the corresponding assembly code (Middle) with some parts omitted for brevity and the corresponding hexadecimal mode of the binary code (Right).

3 Code Action Network for the Function Identification Problem

3.1 Key Idea

In what follows, we present the key idea of our CAN. In a binary, there are external gaps between functions as well as internal gaps inside a non-contiguous function. The external gaps might contain data, jump tables or padding-instruction bytes which do not belong to any function (e.g., additional instructions generated by a compiler such as *nop, int3*). The internal gaps in general might contain data, jump tables or instructions from other functions (e.g., nested functions). We further assume that the internal gaps do not contain any instruction from other functions. It means that if there exist functions nested in a function, our CAN ignores these internal functions. However, we believe that the nested functions are extremely rare in real-world binaries. For example, in the experimental dataset, we observe that there are only 506 nested functions over the total of 757,125 functions (i.e., the occurrence rate is 0.067%).

The key idea of CAN is to encode the task of function scope identification to a sequence of three action states NI (i.e., next inclusion), NE (i.e., next exclusion), and FE (i.e., function end). With the aforementioned assumption, the binaries of interest consist of several functions and the functions in a binary do not intermingle, that is, each function only contains its machine instructions, data, or jump-tables and do not contain any machine instruction of other functions. Each function can be therefore viewed as a collection of bytes where each byte is from a machine instruction of this function (i.e., instruction byte) or data/jump-tables inside this function (i.e., non-instruction byte). To clarify how to proceed over a binary function given a sequence of action states, let us imagine this

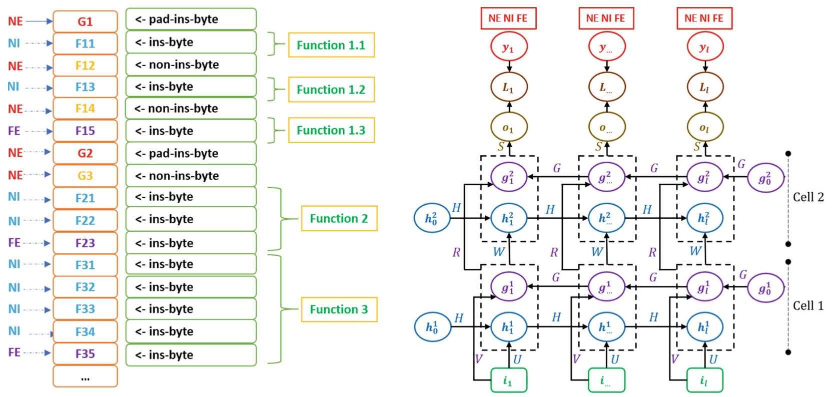


Fig. 2. (The left-hand figure) The key idea of Code Action Network. Assume that we have a sequence of instruction bytes in three functions where the functions may not be contiguous and there exist gaps between the functions. The Code Action Network transforms this sequence of instruction bytes to those of action states (i.e., NI, NE, and FE). (The right-hand figure) The architecture of the Code Action Network. Each output value takes one of three action states *NI*, *NE*, or *FE.* The Code Action Network will learn to map the input sequences of items $(\mathbf{i}_1, \mathbf{i}_2, ..., \mathbf{i}_l)$ to the target output sequence $(\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_l)$ with the loss L_i at each time step t. The h represents for the forward-propagated hidden state (toward the right) while the g stands for the backward-propagated hidden state (toward the left). At each time step t, the predicted output $\mathbf{o}_t$ can benefit from the relevant information of the past from its h and the future from its g.

binary program including many instruction and non-instruction bytes as a tape of many cells wherein each cell contains a instruction or non-instruction byte and a pointer firstly points to the first cell in the tape. The action state NI includes the current instruction or non-instruction byte in the current cell to the current function and moves the pointer to the next cell (i.e., the next instruction or non-instruction byte). The action state NE excludes the current instruction or non-instruction byte in the current cell from the current function and moves the pointer to the next cell. The action state FE counts the current instruction or non-instruction byte in the current cell, ends the current function, starts reading a new function, and moves the pointer to the next cell.

To further explain how to transform a binary program to a sequence of action states, we consider an example binary code depicted in Fig. 2 (the left-hand figure). Assume that we have a sequence of instruction and non-instruction bytes, which belong to *Function 1*, *Function 2* and *Function 3*, respectively where the functions may be not contiguous and there exist gaps between the functions (e.g., the gap between *Function 1* and *Function 2* includes the padding-instruction byte (pad-ins-byte) G2 and the non-instruction (non-ins-byte) byte G3). The pointer of CAN firstly points to G1, labels this padding-instruction byte (pad-ins-byte) as NE since G1 does not belong to any function, and moves to the instruction byte F11. The instruction byte F11 is labeled as NI since it belongs to the function *Function 1.* The pointer then moves to the non-instruction byte F12 which can come from a jump-table or data and labels it as NE because F12 does not belong to any function. After that, the pointer moves to the instruction byte F13 and the non-instruction byte F14 subsequently. F13 and F14 are then labeled as NI and NE respectively since F13 belong to the function *Function 1* while F14 does not belong to any function, and the pointer moves to the instruction byte F15 and labels it as FE since it is the end of the function *Function 1* and we need to start reading the new function (i.e., the function *Function 2*). The pointer subsequently moves to the instruction byte G2 and the non-instruction G3 which can come from a jump-table or data and labels them as NE since they do not belong to any function. The pointer then traverses across the instruction bytes F21, F22, F23 and labels them as NI, NI, FE. The pointer now starts reading the new function (i.e., the function *Function 3*). This process is repeated until the pointer reaches the last instruction or non-instruction byte and we eventually identify all functions.

It is worth noting that if binaries can be disassembled and a function in these binaries can be thus viewed as a collection of instructions and non-instructions, we can perform the aforementioned idea at the machine instruction level wherein each cell in the tape represents an instruction or non-instruction of a binary. The advantages of performing the task of function identification at the machine instruction level include: i) the sequence length of the bidirectional RNN is significantly reduced and ii) the semantic relationship among bytes in a machine instruction and machine instructions can be further exploited. As a consequence, the gradient exploding and vanishing which often occur with long RNNs can be avoided and the model is easier to train while obtaining higher predictive performance and much shorter training times as shown in our experiments.

3.2 Preprocess Input Statement

Byte Level and Machine Instruction Level. To process data for the byte level, we simply take the raw bytes in the text segment of the given binary and input them to CAN-B. To process data for the machine instruction level, we first use Capstone[1] to disassemble the binaries and preprocess the machine instructions obtained from the text segment of a binary before inputting them to CAN-M. This preprocessing step aims to work out fixed length inputs from machine instructions. For each machine instruction, we employ Capstone to detect entire machine instructions, then eliminate redundant prefixes to obtain core parts that contain the opcode and other significant information (see our Supplementary Material for details, available at https://app.box.com/s/iq9u8r).

3.3 Code Action Network Architecture

Training Procedure. The Code Action Network (CAN) is a multicell bidirectional RNN whose architecture is depicted in Fig. 2 (the right-hand figure) where we assume the number of cells over the input is 2. Our CAN takes a binary program $\mathbf{B} = (\mathbf{i}_1, \mathbf{i}_2, \ldots, \mathbf{i}_l)$ including l instructions (non-instructions) for CAN-M or instruction bytes (non-instruction bytes) for CAN-B and learns to output the corresponding sequence of action states $\mathbf{Y} = (\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_l)$ where each $\mathbf{y}_k$ takes one of three action states *NI* (i.e., $\mathbf{y}_k = 1$), *NE* (i.e., $\mathbf{y}_k = 2$), or *FE*(i.e., $\mathbf{y}_k = 3$). The computational process of CAN is as follows:

$$\mathbf{h}_k^1 = \tanh(H^\top \mathbf{h}_{k-1}^1 + U^\top \mathbf{i}_k);\ \mathbf{g}_k^1 = \tanh(G^\top \mathbf{g}_{k+1}^1 + V^\top \mathbf{i}_k);\ \mathbf{h}_k^2 = \tanh(H^\top \mathbf{h}_{k-1}^2 + W^\top [\begin{smallmatrix}\mathbf{h}_k^1\\ \mathbf{g}_k^1\end{smallmatrix}])$$

$$\mathbf{g}_k^2 = \tanh(G^\top \mathbf{g}_{k+1}^2 + R^\top [\begin{smallmatrix}\mathbf{h}_k^1\\ \mathbf{g}_k^1\end{smallmatrix}]);\ \mathbf{o}_k = S^\top [\begin{smallmatrix}\mathbf{h}_k^2\\ \mathbf{g}_k^2\end{smallmatrix}];\ \mathbf{p}_k = \text{softmax}(\mathbf{o}_k)$$

where $k = 1, ...l$, $\mathbf{h}_0^1$, $\mathbf{h}_0^2$, $\mathbf{g}_{l+1}^1 = \mathbf{g}_0^1$, $\mathbf{g}_{l+1}^2 = \mathbf{g}_0^2$ are initial hidden states and $\theta = (U, V, W, H, G, R, S)$ is the model. We further note that $\mathbf{p}_k$, $k = 1, \ldots, l$ is a *discrete* distribution over the three labels NI, NE, and FE.

To find the best model θ^*, we need to solve the following optimization problem:

$$\max_{\theta} \sum_{(\mathbf{B},\mathbf{Y})\in\mathcal{D}} \log p(\mathbf{Y} \mid \mathbf{B}) \tag{1}$$

where $\mathcal{D}$ is the training set including pairs $(\mathbf{B}, \mathbf{Y})$ of the binaries and their corresponding sequence of action states.

Because o_k is a function (lossy summary) of $\mathbf{i}_{1:l}$, we further derive $\log p(\mathbf{Y} \mid \mathbf{B})$ as:

$$\log p(\mathbf{Y} \mid \mathbf{B}) = \sum_{k=1}^{l} \log p(\mathbf{y}_k \mid \mathbf{y}_{1:k-1}, \mathbf{i}_{1:l}) = \sum_{k=1}^{l} \log p(\mathbf{y}_k \mid \mathbf{o}_k)$$

[1] www.capstone-engine.org.

Substituting back to the optimization problem in Eq. (1), we arrive the following optimization problem:

$$\max_{\theta} \sum_{(\mathbf{B},\mathbf{Y})\in\mathcal{D}} \sum_{k=1}^{l} \log p\left(\mathbf{y}_k \mid \mathbf{o}_k\right)$$

where $p(\mathbf{y}_k \mid \mathbf{o}_k)$ is the $\mathbf{y}_k$-th element of the discrete distribution $\mathbf{p}_k$ or in other words, we have $p(\mathbf{y}_k \mid \mathbf{o}_k) = \mathbf{p}_{k,\mathbf{y}_k}$.

Testing Procedure. In what follows, we present how to work out the function scopes in a binary using a trained CAN. The machine instructions/non-instructions for CAN-M or instruction/non-instruction bytes for CAN-B in the testing binary are fed to the trained model to work out the predicted sequence of action states. This predicted sequence of action states is then decoded to the function scopes inside the binary. As shown in Fig. 3, the binary in Fig. 2 when inputted to the trained CAN outputs the sequence of action states NE, NI, ..., NI, FE and is later decoded to the scopes of the functions *Function 1, Function 2* and *Function 3*.

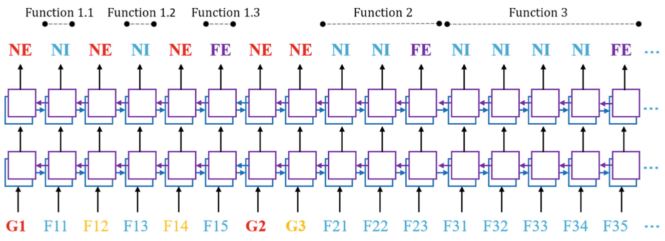

Function 1.1
Function 1.2
Function 1.3
Function 2
Function 3
NE NI NE NI NE FE NE NE NI NI FE NI NI NI NI FE ...
G1 F11 F12 F13 F14 F15 G2 G3 F21 F22 F23 F31 F32 F33 F34 F35 ...

Fig. 3. The testing procedure of our Code Action Network. The sequence of machine instructions/non-instructions or instruction bytes/non-instruction bytes in a binary program is fed to the trained Code Action Network to work out the sequence of action states. Subsequently, the sequence of action states is decoded to the set of functions in this binary.

4 Experiments

In this section, firstly, we present the experimental results of our proposed Code Action Network for the machine instruction level (CAN-M) and the byte level (CAN-B) compared with other baselines including IDA, ByteWeight (BW) no-RFCR, ByteWeight (BW) [2], the Bidirectional RNN (BRNN) [12] and Nucleus [1]. Secondly, we perform error analysis to qualitatively investigate our proposed methods. We also investigate the model behaviour of our CAN-M with various RNN cells and with different size for hidden states (see in our Supplementary Material, available at https://app.box.com/s/iq9u8r).

4.1 Experimental Dataset

We used the dataset from [2,12], which consists of 2,200 different binaries including 2,064 binaries obtained from the *findutils*, *binutils*, and *coreutils* packages and compiled with both *icc* and *gcc* for Linux at four optimization levels O0, O1, O2, and O3. The remaining binaries for Windows are from various well-known open-source projects which were compiled with Microsoft Visual Studio for the x86 (32 bit) and the x86-64 (64 bit) architectures at four optimization levels Od, O1, O2, and Ox.

4.2 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 CAN-M, we used a sequence of 250 hidden states for the x86 architecture and 125 hidden states for the x86-64 architecture where the size of hidden states is 256. For CAN-B, akin to the Bidirectional RNN in [12], we used a sequence length of 1,000 hidden states for the x86 and x86-64 architectures. We employed the Adam optimizer with the default learning rate 0.001 and the mini-batch size of 32. In addition, we applied gradient clipping regularization to prevent the over-fitting problem when training the model. We implemented the Code Action Networks in Python using Tensorflow, an open-source software library for Machine Intelligence developed by the Google Brain Team.

4.3 Experimental Results

Code Action Network Versus Baselines. We compared our CAN-M and CAN-B using the Long Short Term Memory (LSTM) cell and the hidden size of 256 with IDA, the Bidirectional RNN (BRNN), ByteWeight (BW) no-RFCR and ByteWeight (BW) in the task of function start, function end, function boundary and function scope identification. For the well-known tool IDA as well as the Bidirectional RNN, ByteWeight no-RFCR, and ByteWeight methods, we reported the experimental results presented in [2] and [12]. Obviously, the task of function scope identification wherein we need to specify addresses of machine instructions in each function is harder than that of function boundary identification. To compute the function scope results, given a predicted function by CAN variants, we considered their start and end instructions for CAN-M and start and end bytes for CAN-B, and then evaluated measures (e.g., Precision, Recall, and F1 score) based on this pair. In addition, in the function scope identification task, a pair is counted as a correct pair if all predicted bytes or machine instructions accompanied with this pair forms a function that exactly matches to a valid function in the ground truth. In contrast, in the function boundary identification task, we only require the start and end positions of this pair to be correct.

The experimental results in Table 1 show that our proposed CAN-M and CAN-B achieved better predictive performances (i.e., Recall, Precision, and F1 score) compared with the baselines in most cases (PE x86, PE x86-64, ELF x86 and ELF x86-64). For the function boundary identification task, our CAN-B and CAN-M significantly outperformed the baselines in all measures, especially for CAN-M. Interestingly, the predictive performance of our proposed methods on the harder task of function scope identification was higher or comparable with that of the baselines on the easier task of function boundary identification. In comparison with the Bidirectional RNN proposed in [12], our proposed methods slightly outperform it on the function start and function end identification tasks, but significantly surpass this method on the function boundary identification task - the more important task. This result demonstrates the capacity of our methods in efficiently utilizing the contextual relationship carried in consecutive machine instructions or bytes to properly match the function start and end entries for this task. Regarding the amount of time taken for training, our CAN-M took approximately 3,490 s for training in 20,000 iterations, while our CAN-B and the Bidirectional RNN using the same number of iterations with the sequence length 1,000 took about 12,030 seconds (i.e., roughly four times slower). This is due to a much smaller sequence length of CAN-M compared with CAN-B and the Bidirectional RNN.

Code Action Network Versus Bidirectional RNN, ByteWeight and Nucleus. We also compared the average predictive performance for case by case including the function start, function bound and function scope identifications of our CAN-M and CAN-B using the hidden size of 256 and LSTM cell with the Bidirectional RNN, ByteWeight, and Nucleus in both Linux and Windows platforms. For Nucleus [1], we reported the experimental results reported in that paper. The experimental results in Table 2 indicate that our CAN-M and CAN-B again outperformed the baselines, while CAN-M obtained the highest predictive performances in all measures (Recall, Precision and F1 score).

4.4 Error Analysis

For a qualitative assessment, we performed error analysis of our CAN-M and CAN-B for all cases including PEx86, PEx64, ELFx86 and ELFx64.

At the machine instruction level, we observed that there are *4,714, 4,464, 3,320* and *8,147* different types of machine instructions for function start while there are *1,926, 5,523, 9,082* and *11,421* different types of machine instructions for function end in the PEx86, PEx64, ELFx86 and ELFx64 datasets respectively. At byte level, we found that there are *91, 49, 41* and *53* different types of instruction bytes for function start while there are *166, 125, 133* and *126* different types of bytes for function end in the PEx86, PEx64, ELFx86 and ELFx64 datasets respectively. Obviously, these diverse ranges in the function start and function styles make the task of function identification really challenging. In all four cases (PEx86, PEx64, ELFx86 and ELFx64), the compilers in use often add padding between functions such as *nop*, *int3*.

We summarize some observations for our methods performance as follows:

– Shin et al. [12] and Bao et al. [2] commonly mentioned that for the functions that contain either several function start or function end styles inside, their models tend to confuse in determining the true start or end points, hence offering many false positives. This is due to a high level of ambiguity in the start or end entries for these functions. However, it is not the case for our proposed CAN-M and CAN-B. For example, at the machine instruction level with PE x86, we found that the functions which contain more than one function start style or function end style account for 98.38% and 28.00% of the testing set and when predicting these functions, our proposed CAN-M has 0.28% false negative rate and 0.24% false positive rate as well as 1.56% false negative rate and 1.09% false positive rate.
– Our proposed methods also share the same behavior as the method in [12] in predicting some first and last items in an input sequence, that is, the CAN-M and CAN-B sometimes offer false positives and negatives when predicting some first and last instructions or bytes in an input sequence. More specifically, if an input sequence involves several functions, the start of the first function and the end of the last function are more likely to be predicted incorrectly. This is possibly due to the scarcity of context before or after them. For example, at the machine instruction level with PE x86, we record that there is about 2.39% of input sequences which contain function ends at some first and last input items. When predicting these function end entries, our proposed CAN-M obtains 21.21% false positive rate and 27.27% false negative rate.

Table 1. Comparison of our Code Action Network and baselines (Best in **bold**, second best in underline). Noting that f.s, f.e, f.b and f.sc stand for func. start, func. end, func. boundary and func. scope while R, P, and F1 represent Recall, Precision and F1 score respectively.

Task	**Architectures**	**ELF x86**			**ELF x86-64**			**PE x86**			**PE x86-64**		
	Methods	**R**	**P**	**F1**	**R**	**P**	**F1**	**R**	**P**	**F1**	**R**	**P**	**F1**
(f.s)	IDA	58.34%	70.97%	64.04%	55.50%	74.20%	63.50%	87.80%	94.67%	91.11%	93.34%	98.22%	95.72%
	BW no-RFCR	96.17%	98.36%	97.25%	97.57%	99.11%	98.33%	92.13%	96.75%	94.38%	96.22%	97.74%	96.97%
	BW	97.94%	98.41%	98.17%	**98.47%**	99.14%	98.80%	95.37%	93.78%	94.57%	97.98%	97.88%	97.93%
	BRNN	99.06%	*99.56%*	99.31%	97.80%	98.80%	98.30%	98.46%	99.01%	98.73%	*99.09%*	*99.52%*	*99.30%*
	CAN-B	*99.23%*	99.41%	*99.32%*	*98.19%*	*99.05%*	*98.62%*	*98.95%*	*99.53%*	*99.24%*	**99.20%**	99.46%	**99.33%**
	CAN-M	**99.35%**	**99.61%**	**99.48%**	98.02%	**99.34%**	**98.68%**	**99.52%**	**99.67%**	**99.59%**	99.05%	***99.53%***	99.29%
(f.e)	BRNN	97.87%	98.69%	98.28%	95.03%	97.45%	96.22%	98.35%	99.24%	98.79%	**99.20%**	99.28%	**99.24%**
	CAN-B	*99.16%*	*99.38%*	*99.27%*	**98.34%**	*99.20%*	**98.77%**	*98.82%*	*99.39%*	*99.10%*	*99.15%*	*99.30%*	*99.22%*
	CAN-M	**99.30%**	**99.56%**	**99.43%**	*97.97%*	**99.29%**	*98.63%*	**99.56%**	**99.71%**	**99.64%**	99.12%	**99.31%**	99.21%
(f.b)	IDA	56.53%	70.63%	62.80%	53.46%	72.84%	61.66%	87.10%	93.93%	90.39%	93.24%	98.11%	95.61%
	BW no-RFCR	90.58%	92.85%	91.70%	91.59%	93.17%	92.37%	90.48%	95.03%	92.70%	91.35%	92.87%	92.10%
	BW	92.29%	92.78%	92.53%	92.52%	93.22%	92.87%	93.91%	92.30%	93.10%	93.13%	93.04%	93.08%
	BRNN	95.34%	97.75%	96.53%	89.91%	94.85%	92.31%	95.27%	97.53%	96.39%	97.33%	**98.43%**	97.88%
	CAN-B	*98.08%*	*98.29%*	*98.18%*	**96.45%**	*97.24%*	**96.84%**	*97.81%*	*98.36%*	*98.08%*	**97.89%**	98.27%	**98.08%**
	CAN-M	**98.43%**	**98.68%**	**98.55%**	*96.13%*	**97.34%**	*96.73%*	**98.99%**	**99.14%**	**99.06%**	*97.63%*	*98.39%*	*98.01%*
(f.sc)	**CAN-B**	*98.03%*	*98.25%*	*98.14%*	**96.28%**	*97.10%*	**96.69%**	*97.75%*	*98.31%*	*98.03%*	**97.83%**	*98.22%*	**98.02%**
	CAN-M	**98.40%**	**98.65%**	**98.52%**	*95.94%*	**97.21%**	*96.57%*	**98.97%**	**99.12%**	**99.05%**	*97.52%*	**98.28%**	*97.90%*

Table 2. Comparison with the baselines (the Bidirectional RNN, ByteWeight and Nucleus) using average scores for all architectures (x86 and x86-64) for both Linux and Windows of our Code Action Network. The experimental results for Nucleus are from the original paper using the same dataset (Best performance in **bold**, second best in underline).

Tasks	**Function Start**			**Function Bound**			**Function Scope**		
Methods	**Recall**	**Precision**	**F1**	**Recall**	**Precision**	**F1**	**Recall**	**Precision**	**F1**
Nucleus	94%	96%	94.99%	88%	96%	91.83%	88%	96%	91.83%
ByteWeight	97.44%	97.30%	97.37%	92.96%	92.84%	92.90%	-	-	-
Bidirectional RNN	98.60%	99.22%	98.92%	94.46%	97.14%	95.78%	-	-	-
CAN-B	*98.89%*	*99.36%*	*99.12%*	*97.56%*	*98.04%*	*97.80%*	*97.47%*	*97.97%*	*97.72%*
CAN-M	**98.99%**	**99.54%**	**99.26%**	**97.80%**	**98.39%**	**98.09%**	**97.71%**	**98.32%**	**98.01%**

5 Conclusion

In this paper, we have proposed the novel Code Action Network (CAN) 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 CAN leverages the underlying idea of a multicell bidirectional recurrent neural network with the idea of encoding the task of function scope identification to a sequence of three action states NI (i.e., next inclusion), NE (i.e., next exclusion), and FE (i.e., function end) in order to tackle function scope identification, the hardest and most crucial task in function identification. The experimental results show that the CAN can achieve state-of-the-art performance in terms of efficiency and efficacy.

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

References

1. Andriesse, D., Slowinska, A., Bos, H.: Compiler-agnostic function detection in binaries. In: IEEE European Symposium on Security and Privacy (EuroS&P) (2017)
2. Bao, T., Burket, J., Woo, M.: Byteweight: learning to recognize functions in binary code. In: 23rd USENIX Security Symposium (USENIX Security 2014) (2014)
3. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems 25 (2012)
4. Kruegel, C., Robertson, W., Valeur, F., Vigna, G.: Static disassembly of obfuscated binaries. In: Proceedings of Conference on USENIX Security Symposium (2004)
5. Laurenzano, M.A., Tikir, M.M., Carrington, L., Snavely, A.: PEBIL: efficient static binary instrumentation for Linux. In: International Symposium on Performance Analysis of Systems and Software (ISPASS) (2010)
6. Le, T., et al.: Maximal divergence sequential autoencoder for binary software vulnerability detection. In: International Conference on Learning Representations (2019)

7. Nguyen, T., et al.: Deep cost-sensitive kernel machine for binary software vulnerability detection. In: Pacific-Asia Conference on Knowledge Discovery and Data Mining (2020)
8. Nguyen, V., et al.: Deep domain adaptation for vulnerable code function identification. In: International Joint Conference on Neural Networks (2019)
9. Perkins, J.H., et al.: Automatically patching errors in deployed software. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (2009)
10. Pewny, J., Garmany, B., Gawlik, R., Rossow, C., Holz, T.: Cross-architecture bug search in binary executables. In: Proceedings of IEEE Symposium on Security and Privacy (2015)
11. Rosenblum, N.E., Zhu, X., Miller, B.P., Hunt, K.: Learning to analyze binary computer code. In: AAAI, pp. 798–804 (2008)
12. Shin, E.C.R., Song, D., Moazzezi, R.: Recognizing functions in binaries with neural networks. In: 24th USENIX Security Symposium (USENIX Security 2015) (2015)
13. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Proceedings of the 27th International Conference on Neural Information Processing Systems, vol. 2 (2014)
14. Zhang, M., Sekar, R.: Control flow integrity for COTS binaries. In: Proceedings of the 22nd USENIX Conference on Security (2013)