Recognizing Functions in Binaries with Neural Networks

Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi

UC Berkeley

Key Contribution

- Recurrent Neural Networks (RNNs) can solve the function identification problem more efficiently and accurately than previous state-of-the-art ML and traditional methods

Outline

- The problem: function identification in stripped binaries
- Previous solutions and their inadequacies; why RNN?
- Network architecture and design decisions
- Evaluation and limitations
- Key takeaways

Ultra quick refresher on stripped binaries

- Source code to execution:

  $\underbrace{\text{Preprocessing -> Compiling -> Assembly -> Linking}}_{\text{Compilation}}$ -> Loading

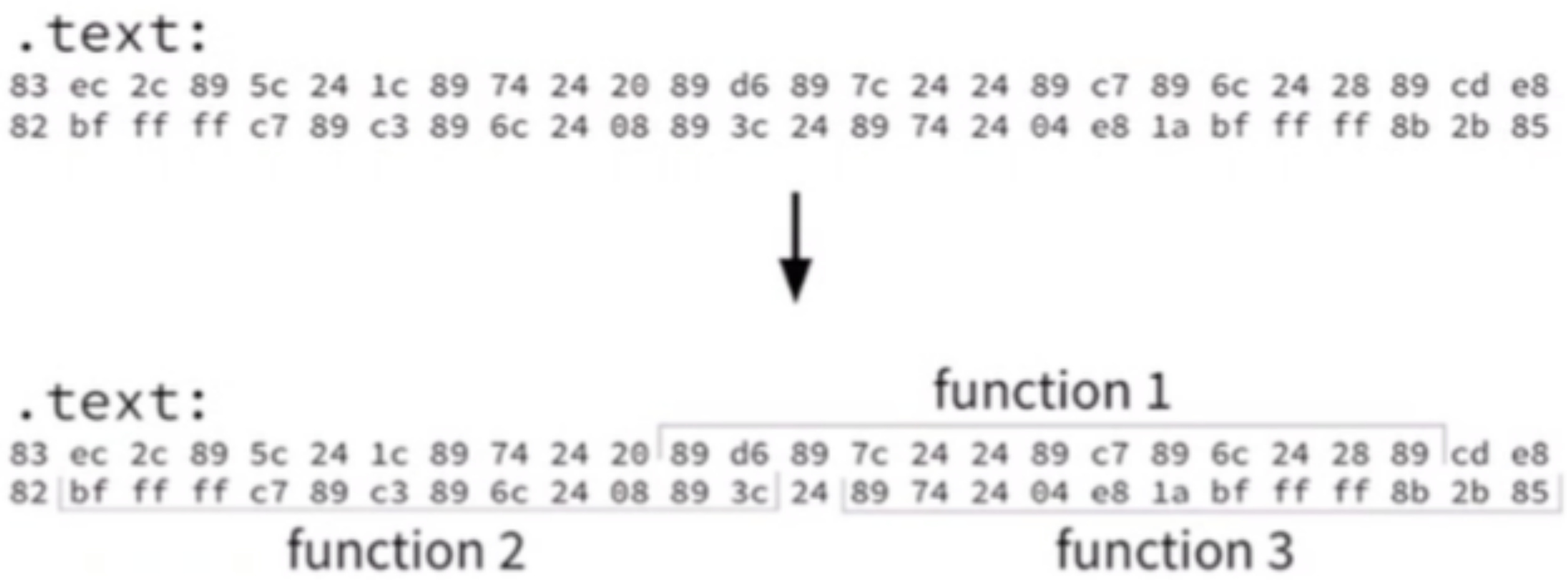
- Symbol table:

  Data structure used during compilation that maps identifiers from the source code to their type info and memory addresses

- A stripped binary is an executable whose symbol table is removed

Function Identification

- Given a stripped binary executable, we want to identify the start and end bytes of each function in the binary



Why do we care?

- Malware analysis
- Debugging
- Decompiling
- Retrofitting control-flow integrity
- Binary rewriting

Why is this difficult?

- During compilation the assembler strips away function symbols, so we must make deductions based on incomplete information
- Different compilers and optimization settings generate different code
- Disassembly is hard because x86 uses varying length instructions

Compiler generated code can vary

```c
#include <stdio.h>

int add(int x, int y) { return x + y; }

int main(int argc, char **argv)
{
    int x = 3;
    int y = 5;
    int z = add(x, y);

    printf("%d\n", z);

    return 0;
}
```

Compiler generated code can vary

```c
int add(int x, int y) { return x + y; }
```

Source code

```asm
    .globl  add
    .type   add, @function
add:
    push    ebp
    mov ebp, esp
    call    __x86.get_pc_thunk.ax
    add eax, OFFSET FLAT:_GLOBAL_OFFSET_TABLE_
    mov edx, DWORD PTR 8[ebp]
    mov eax, DWORD PTR 12[ebp]
    add eax, edx
    pop ebp
    ret
```

Compiled with gcc **-O0** -S -fno-asynchronous-unwind-tables

```asm
    .globl  add
    .type   add, @function
add:
    mov eax, DWORD PTR 8[esp]
    add eax, DWORD PTR 4[esp]
    ret
```

Compiled with gcc **-O3** -S -fno-asynchronous-unwind-tables

Disassembly is hard

- x86 uses varying length instructions; depending on which byte disassembly begins at the instructions can be interpreted differently
- Data is often mixed in code, e.g. jump tables
- Adversaries can use many anti-disassembly techniques to throw off disassemblers

Disassembly is hard

Anti-disassembly example: Jumping over a rogue byte

(not important to remainder of presentation, feel free to ignore)

```
                jmp     short near ptr loc_2+1
; ---------------------------------------------------------------
loc_2:                              ; CODE XREF: seg000:00000000j
                call    near ptr 15FF2A71h ❶
                or      [ecx], dl
                inc     eax
; ---------------------------------------------------------------
                db    0
```

```
                jmp     short loc_3
; ---------------------------------------------------------------
                db 0E8h
; ---------------------------------------------------------------
loc_3:                              ; CODE XREF: seg000:00000000j
                push    2Ah
                call    Sleep ❶
```

Notation

The input is code C, a sequence of bytes $C[0], C[1], \dots, C[l]$ where $C[i] \in \mathbb{Z}_{256}$ is the i^{th} byte in the sequence

The n functions in the code are denoted $f_1, f_2, \dots, f_n$, and the bytes belonging to function f_i are denoted $f_{i,1}, f_{i,2}, \dots, f_{i,l_i}$ where l_i is the total number of bytes in f_i

Formal Task Definition

- Function boundary identification:

  Given C, find $\{(f_{1,1}, f_{1,l_1}), (f_{2,1}, f_{2,l_2}), \dots, (f_{n,1}, f_{n,l_n})\}$

- Easier subtasks- function start/end identification:

  Given C, find $\{f_{1,1}, f_{2,1}, \dots, f_{n,1}\}$

  Given C, find $\{f_{1,l_1}, f_{2,l_2}, \dots, f_{n,l_n}\}$

Outline

- The problem: function identification in stripped binaries
- Previous solutions and their inadequacies; why RNN?
- Network architecture and design decisions
- Evaluation and limitations
- Key takeaways

Traditional approach

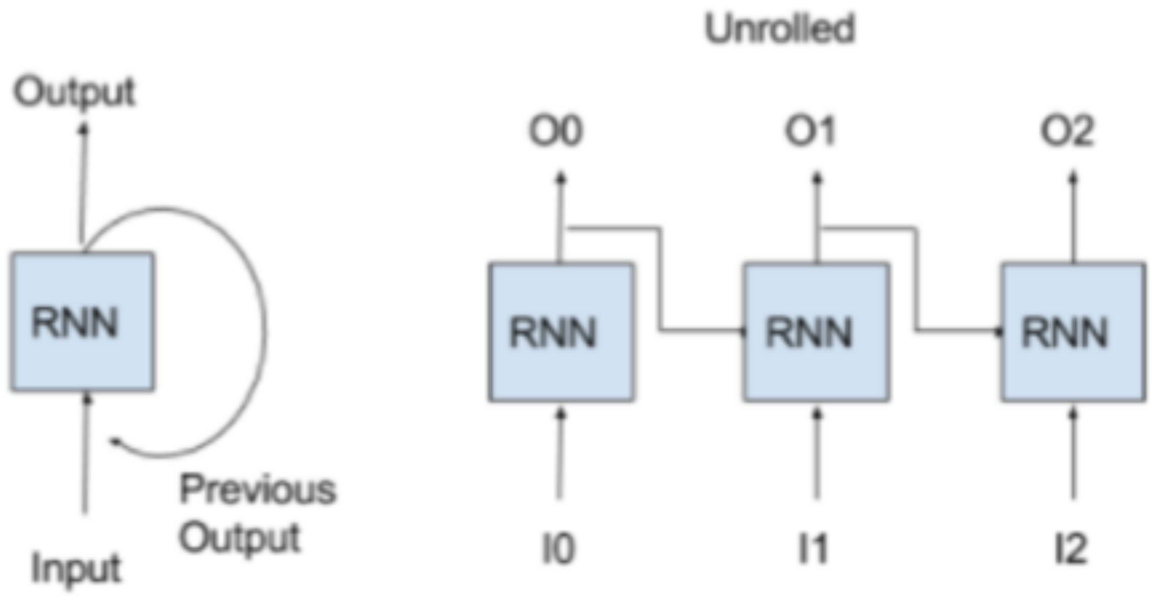
- Disassemble machine code into assembly, then identify functions with code references and pattern matching against manually curated function prologue/epilogue signatures
- Used by popular commercial tools: IDA Pro/Hex-Rays, Phoenix, Boomerang etc.
- Fast but inaccurate: Bao et al. showed that the even most accurate tool, IDA Pro, had a 41.81% true positives, 21.38% false negatives and 36.81% false positives on a test set of ~1 million functions

Machine learning approach: ByteWeight

- Machine Learning based approach, uses weighted prefix trees to learn function prologues from data
- Requires preprocessing by disassembler; works on assembly code
- Good accuracy but at the cost of efficiency: 92%+ F1 score on Windows and Linux binaries, but 587 hours to train on a training set of 2,200 binaries

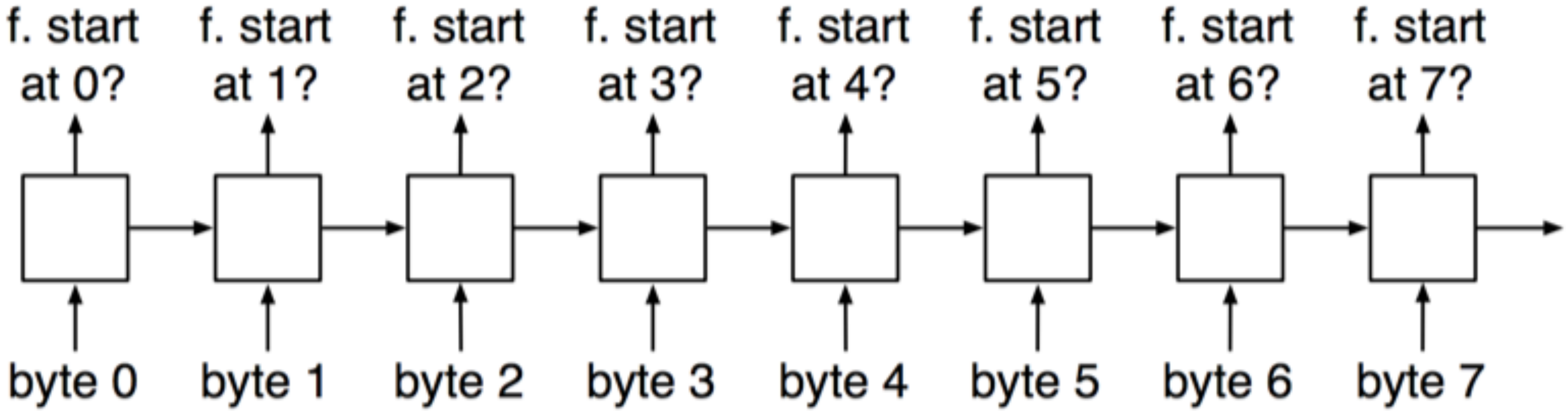
Review of RNNs

- Good for processing sequence data, widely used in NLP
- Maintains state while iterating through sequence elements



Why RNNs are a good fit

- Essentially, our task can be formulated as iterating through a sequence of bytes, and identifying the bytes that represent the start or end of a function



Outline

- The problem: function identification in stripped binaries
- Previous solutions and their inadequacies; why RNN?
- Network architecture and design decisions
- Evaluation and limitations
- Key takeaways

Dataset

- 2200 Linux/Windows binaries compiled with GCC, ICC, and Visual Studio under 4 different optimization levels
- Same dataset as ByteWeight; enables direct comparison

	ELF x86	ELF x86-64	PE x86	PE x86-64
Number of binaries	1,032	1,032	68	68
Number of bytes	138,547,936	145,544,012	29,093,888	33,351,168
Number of functions	303,238	295,121	93,288	94,548
Average function length	448.84	499.54	292.85	330.03

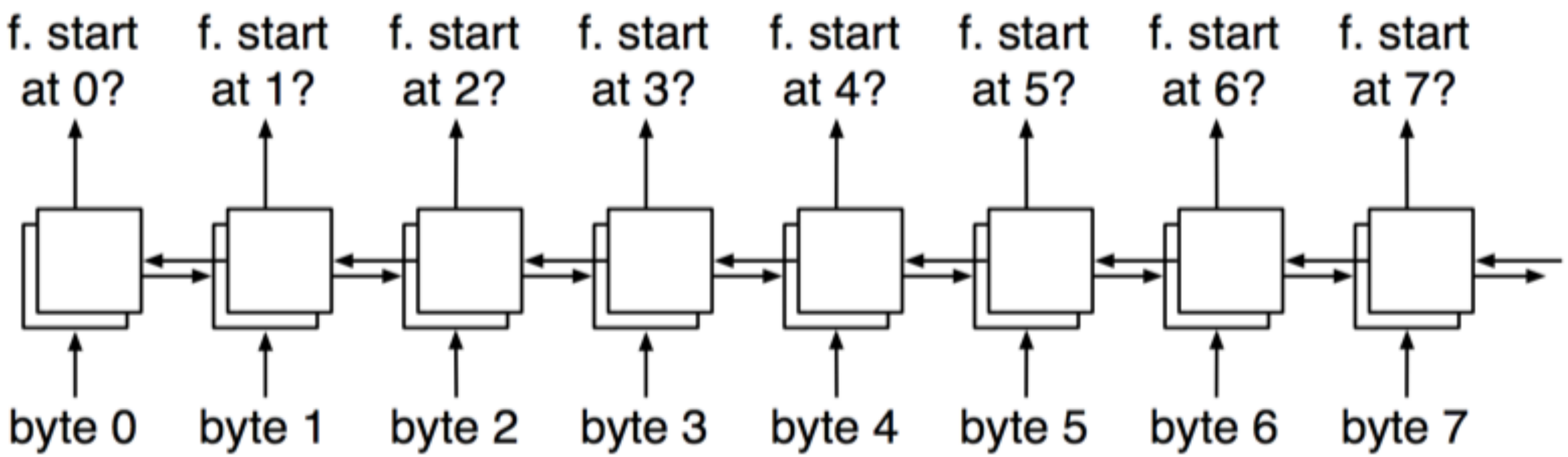
Data Preparation

- Ignore all binary data except for the .text section which contains the actual machine code instructions
- Extract 100,000 1000-byte chunks from the 2200 binaries to build training set
- Encode each byte with one-hot encoding to an $\mathbb{R}^{256}$ vector
- No disassembly required!
- Authors mention code references could be used to increase accuracy, but did not attempt this due to complexity

Bi-directional RNNs

- Uni-directional RNNs don’t take advantage of sequence elements that are later in the sequence than the current element
- As a result, the network must make its classification while only looking at bytes that come before the current byte
- This restriction is necessary for many sequence data classification tasks, but not for function identification- complete sequences are always available

Bi-directional RNNs



Architecture and Hyperparameters

- Bi-directional RNN
- One hidden layer with 16 bi-directional RNN nodes
- Softmax layer: function start; function end; neither
- Mini-batch gradient descent using RMSprop, batch size 32

Architecture and Hyperparameters

- 10-fold cross validation with 10% of training set to tune hyperparameters

	Function start identification				Function end identification			
	ELF x86	ELF x86-64	PE x86	PE x86-64	ELF x86	ELF x86-64	PE x86	PE x86-64
Separate								
$h = 8, l = 1$	98.88%	96.07%	98.04%	99.42%	95.93%	92.94%	97.98%	99.25%
$h = 8, l = 2$	99.03%	97.69%	98.00%	99.43%	97.71%	94.49%	98.30%	99.19%
$h = 16, l = 1$	99.24%	98.13%	98.33%	99.50%	98.09%	95.74%	98.56%	99.24%
Shared								
$h = 8, l = 1$	97.79%	95.28%	97.30%	99.23%	95.86%	91.94%	97.08%	98.90%
$h = 8, l = 2$	98.60%	96.67%	97.96%	99.45%	97.41%	94.92%	97.58%	99.12%
$h = 16, l = 1$	98.29%	97.41%	98.42%	99.47%	97.20%	95.51%	98.32%	99.38%

Outline

- The problem: function identification in stripped binaries
- Previous solutions and their inadequacies; why RNN?
- Network architecture and design decisions
- Evaluation and limitations
- Key takeaways

Evaluation Metrics

- Network performance: precision, recall, F1 score (harmonic mean of precision and recall)

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

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

$$\text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

- Efficiency: computational power consumed by training

Evaluation: Start/End Identification

	ELF x86			ELF x86-64		
	P	R	F1	P	R	F1
ByteWeight (func. start)	98.41%	97.94%	98.17%	99.14%	98.47%	98.80%
Our models (func. start)	99.56%	99.06%	99.31%	98.80%	97.80%	98.30%
Our models (func. end)	98.69%	97.87%	98.28%	97.45%	95.03%	96.22%
	PE x86			PE x86-64		
	P	R	F1	P	R	F1
ByteWeight (func. start)	93.78%	95.37%	94.57%	97.88%	97.98%	97.93%
Our models (func. start)	99.01%	98.46%	98.74%	99.52%	99.09%	99.31%
Our models (func. end)	99.24%	98.35%	98.79%	99.28%	99.20%	99.24%

Evaluation: Boundary Identification

	ELF x86			ELF x86-64		
	P	R	F1	P	R	F1
ByteWeight	92.78%	92.29%	92.53%	93.22%	92.52%	92.87%
Our models	97.75%	95.34%	96.53%	94.85%	89.91%	92.32%
	PE x86			**PE x86-64**		
	P	R	F1	P	R	F1
ByteWeight	92.30%	93.91%	93.10%	93.04%	93.13%	93.08%
Our models	97.53%	95.27%	96.39%	98.43%	97.33%	97.88%

Evaluation: Training Time

- 7x speed up in training time
- Total training time of ByteWeight: 587 hours
- Total training time of Bi-directional RNN: 80 hours

	ELF x86	ELF x86-64	PE x86	PE x86-64
Our models (func. boundary)	1061.76 s	1017.90 s	236.93 s	264.50 s
ByteWeight (func. start only)	3296.98 s	5718.84 s	10269.19 s	11904.06 s
ByteWeight (func. boundary)	367018.53 s	412223.55 s	54482.30 s	87661.01 s
ByteWeight (func. boundary with RFCR)	457997.09 s	593169.73 s	84602.56 s	97627.44 s

Limitations

- Does not account for adversarial inputs that come from a different distribution than benign training set
- Identification for GCC binaries on x86-64 architecture is less accurate
- ICC will generate functions with multiple entry points as an optimization technique; this causes many false negatives

Key Takeaways

- Function identification in stripped binaries is a binary analysis problem critical to many security domains
- Bi-directional RNNs can solve the function identification problem more efficiently and accurately than previous state-of-the-art ML and traditional methods
- More research needs to be done to increase robustness of function identification against adversarial inputs, which are common for security tasks