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:
  Preprocessing -> Compiling -> Assembly -> Linking -> Loading

   Compilation

• 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

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

```
.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 `-O3` `-S` `-fno-asynchronous-unwind-tables`

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

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

The $n$ functions in the code are denoted $f_1, f_2, \ldots, f_n$, and the bytes belonging to function $f_i$ are denoted $f_{i,1}, f_{i,2}, \ldots, f_{i,l_i}$ 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}), \ldots, (f_{n,1}, f_{n,l_n})\}

• Easier subtasks- function start/end identification:

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

<table>
<thead>
<tr>
<th></th>
<th>Function start identification</th>
<th>Function end identification</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ELF x86</td>
<td>ELF x86-64</td>
</tr>
<tr>
<td>Separate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$h = 8, l = 1$</td>
<td>98.88%</td>
<td>96.07%</td>
</tr>
<tr>
<td>$h = 8, l = 2$</td>
<td>99.03%</td>
<td>97.69%</td>
</tr>
<tr>
<td>$h = 16, l = 1$</td>
<td>99.24%</td>
<td>98.13%</td>
</tr>
<tr>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$h = 8, l = 1$</td>
<td>97.79%</td>
<td>95.28%</td>
</tr>
<tr>
<td>$h = 8, l = 2$</td>
<td>98.60%</td>
<td>96.67%</td>
</tr>
<tr>
<td>$h = 16, l = 1$</td>
<td>98.29%</td>
<td>97.41%</td>
</tr>
</tbody>
</table>
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)

\[
\text{Precision} = \frac{TP}{TP + FP} \\
\text{Recall} = \frac{TP}{TP + FN} \\
F1 = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}
\]

• Efficiency: computational power consumed by training
Evaluation: Start/End Identification

<table>
<thead>
<tr>
<th></th>
<th>ELF x86</th>
<th>ELF x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>R</td>
</tr>
<tr>
<td>ByteWeight (func. start)</td>
<td>98.41%</td>
<td>97.94%</td>
</tr>
<tr>
<td>Our models (func. start)</td>
<td>99.56%</td>
<td>99.06%</td>
</tr>
<tr>
<td>Our models (func. end)</td>
<td>98.69%</td>
<td>97.87%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>PE x86</th>
<th>PE x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>R</td>
</tr>
<tr>
<td>ByteWeight (func. start)</td>
<td>93.78%</td>
<td>95.37%</td>
</tr>
<tr>
<td>Our models (func. start)</td>
<td>99.01%</td>
<td>98.46%</td>
</tr>
<tr>
<td>Our models (func. end)</td>
<td>99.24%</td>
<td>98.35%</td>
</tr>
</tbody>
</table>
Evaluation: Boundary Identification

<table>
<thead>
<tr>
<th></th>
<th>ELF x86</th>
<th>ELF x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>R</td>
</tr>
<tr>
<td>ByteWeight</td>
<td>92.78%</td>
<td>92.29%</td>
</tr>
<tr>
<td>Our models</td>
<td>97.75%</td>
<td>95.34%</td>
</tr>
<tr>
<td></td>
<td>PE x86</td>
<td>PE x86-64</td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>R</td>
</tr>
<tr>
<td>ByteWeight</td>
<td>92.30%</td>
<td>93.91%</td>
</tr>
<tr>
<td>Our models</td>
<td>97.53%</td>
<td>95.27%</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th></th>
<th>ELF x86</th>
<th>ELF x86-64</th>
<th>PE x86</th>
<th>PE x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our models (func. boundary)</td>
<td>1061.76 s</td>
<td>1017.90 s</td>
<td>236.93 s</td>
<td>264.50 s</td>
</tr>
<tr>
<td>ByteWeight (func. start only)</td>
<td>3296.98 s</td>
<td>5718.84 s</td>
<td>10269.19 s</td>
<td>11904.06 s</td>
</tr>
<tr>
<td>ByteWeight (func. boundary)</td>
<td>367018.53 s</td>
<td>412223.55 s</td>
<td>54482.30 s</td>
<td>87661.01 s</td>
</tr>
<tr>
<td>ByteWeight (func. boundary with RFCR)</td>
<td>457997.09 s</td>
<td>593169.73 s</td>
<td>84602.56 s</td>
<td>97627.44 s</td>
</tr>
</tbody>
</table>
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.