Facilitating Postmortem Program Analysis with Deep Learning

Xinyu Xing\textsuperscript{1,2}, Jimmy Su\textsuperscript{1}, Wenbo Guo\textsuperscript{1,2}

1. JD Security Research Center in Silicon Valley
2. The Pennsylvania State University
Grim Reality

• Despite intensive in-house software testing, programs inevitably contain defects.
  • Accidentally terminate or crash at post-deployment stage.
  • Exploited as security loopholes.

• Software debugging is expensive and needs intensive manual efforts.
  • Debugging software cost has risen to $312 billion per year globally [1].
  • Developers spend 50% of their programming time finding and fixing bugs [2].
Postmortem Program Analysis

- Track down the root cause of a software crash at the binary level with out source code.

- Analysis a crash dump and identify the execution path leading to the crash.
  - Reversely execute an instruction trace (starting from the crash site).

- Perform data flow analysis against the identified path.
  - Reconstruct the data flow that a program followed prior to its crash
  - Examine how a bad value was passed to the crash site.
Reverse Execution – Invertible Instructions

- add eax 10
- sub eax 216
- dec eax
- inc eax

- eax-10
- eax+216
- eax+1
- eax-1
Reverse Execution – Non-invertible Instructions

mov eax 0x5
xor ecx ecx
Reverse Execution with Backward Data Flow Analysis

L1. `mov ebx edx`
L2. `mov ecx [edi]`
`ebx == edx` before the execution of L5
L4. `mov ecx 0x77`
L5. `mov ebx [eax]

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x80</td>
</tr>
<tr>
<td>ebx</td>
<td>0x80</td>
</tr>
<tr>
<td>ecx</td>
<td>0x77</td>
</tr>
<tr>
<td>edx</td>
<td>0x32</td>
</tr>
<tr>
<td>edi</td>
<td>0x45</td>
</tr>
<tr>
<td>esi</td>
<td>0x22</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x22</td>
<td>0xE8</td>
</tr>
<tr>
<td>0x45</td>
<td>0x22</td>
</tr>
<tr>
<td>0x77</td>
<td>0xB7</td>
</tr>
<tr>
<td>0x80</td>
<td>0x95</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

10/6/18

JD.com
Challenge – Memory Alias Issue

L1. mov ebx edx
L2. mov ecx [edi]
L3. mov [ecx] esi
L4. mov ecx 0x77
L5. mov ebx [eax]

Wrong!

ecx == [edi] before the execution of L4

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x80</td>
</tr>
<tr>
<td>ebx</td>
<td>0x32</td>
</tr>
<tr>
<td>ecx</td>
<td>0x77</td>
</tr>
<tr>
<td>edx</td>
<td>0x32</td>
</tr>
<tr>
<td>edi</td>
<td>0x45</td>
</tr>
<tr>
<td>esi</td>
<td>0x22</td>
</tr>
<tr>
<td>res</td>
<td>0x22</td>
</tr>
<tr>
<td></td>
<td>0x45</td>
</tr>
<tr>
<td></td>
<td>0x95</td>
</tr>
</tbody>
</table>

Memory Address
0x22 0xE8 0x45 0x22 0x77 0xB7 0x80 0x95 0x32
Challenge – Memory Alias Issue

L1. mov ebx edx
L2. mov ecx [edi]
L3. mov [ecx] esi
L4. mov ecx 0x77
L5. mov ebx [eax]

Pointing to the same memory region

edi == ecx right before the execution of L4

<table>
<thead>
<tr>
<th>Registers</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x80</td>
</tr>
<tr>
<td>ebx</td>
<td>0x95</td>
</tr>
<tr>
<td>ecx</td>
<td>0x77</td>
</tr>
<tr>
<td>edx</td>
<td>0x32</td>
</tr>
<tr>
<td>edi</td>
<td>0x45</td>
</tr>
<tr>
<td>esi</td>
<td>0x22</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>ress</td>
<td>0x22</td>
</tr>
<tr>
<td></td>
<td>0xE8</td>
</tr>
<tr>
<td></td>
<td>0x45</td>
</tr>
<tr>
<td></td>
<td>0x22</td>
</tr>
</tbody>
</table>
Roadmap

• Value-set Analysis.
• Challenges of VSA in postmortem program analysis.
• Design overview.
• Technical details.
• Evaluation in real world crashes.
• Summary.
Value-set Analysis

- State-of-the-art binary level alias analysis technique.
- Being integrated into a variety of binary analysis frameworks (Angr, BAP).
- High level idea:
  - Given a control flow, VSA first assigns each instruction into different memory region (Heap, Stack, global).
  - Tracks down a-locs: register, memory call on stack, heap or global/
  - Compute a value set for each a-loc: (global, stack, heap).
  - Identify memory alias according to the value sets.
Value-set Analysis

- Demonstration of Value-set Analysis.

```
  1    sub    esp,44
  2    lea    eax,[esp+4]
  3    lea    ebx,[esp+24]
  4    mov    [esp+0],eax
  5    mov    ecx,0

L1   mov    edx,[0xC4]
  7    mov    [eax],edx
  8    mov    edx,[0xC8]
  9    mov    [ebx],edx
 10   add    eax,4
 11   add    ebx,4
 12   inc    ecx
 13   cmp    ecx,1
 14   jl     L1
 15   mov    [esp+4],ecx
```

<table>
<thead>
<tr>
<th>Complete Trace</th>
<th>A-loc</th>
<th>Value-set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>esp</td>
<td>(⊥, [-44, -44], ⊥)</td>
</tr>
<tr>
<td>2</td>
<td>eax</td>
<td>(⊥, [-40, -40], ⊥)</td>
</tr>
<tr>
<td>3</td>
<td>ebx</td>
<td>(⊥, [-20, -20], ⊥)</td>
</tr>
<tr>
<td>4</td>
<td>[esp]</td>
<td>(⊥, [-44, -44], ⊥)</td>
</tr>
<tr>
<td>L1</td>
<td></td>
<td>(⊥, [-40, -40], ⊥)</td>
</tr>
<tr>
<td>5</td>
<td>ecx</td>
<td>([0, 0], ⊥, ⊥)</td>
</tr>
<tr>
<td>6</td>
<td>edx</td>
<td>([0, 0], ⊥, ⊥)</td>
</tr>
<tr>
<td>7</td>
<td>[eax]</td>
<td>([0, 0], ⊥, ⊥)</td>
</tr>
<tr>
<td>L1</td>
<td></td>
<td>(⊥, [-40, -40], ⊥)</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>([0, 0], ⊥, ⊥)</td>
</tr>
<tr>
<td>9</td>
<td>[ebx]</td>
<td>([0, 0], ⊥, ⊥)</td>
</tr>
<tr>
<td>10</td>
<td>eax</td>
<td>(⊥, [-36, -36], ⊥)</td>
</tr>
<tr>
<td>11</td>
<td>ebx</td>
<td>(⊥, [-16, -16], ⊥)</td>
</tr>
<tr>
<td>12</td>
<td>ecx</td>
<td>([1, 1], ⊥, ⊥)</td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>[esp+4]</td>
<td>([1, 1], ⊥, ⊥)</td>
</tr>
<tr>
<td></td>
<td>(⊥, [-40, -40], ⊥)</td>
<td></td>
</tr>
</tbody>
</table>
Why not Value-set Analysis in Postmortem Analysis

- In the content of postmortem program analysis:
  - The full control flow or execution trace is not available.
    - Core dump can only record limit length of execution trace.
  - VSA will perform over-approximation in value-set construction.

```
L1  mov    edx,[0xC4]
    mov    [eax],edx
7    mov    edx,[0xC8]
8    mov    [ebx],edx
9    add    eax,4
10    add    ebx,4
12    inc    ecx
13    cmp    ecx,1
14    jl     L1
15    mov    [esp+4],ecx
```
Why not Value-set Analysis in Postmortem Analysis

• Alias Analysis Results.
  • Complete trace: **100%** correctly identify the alias pairs.
  • Incomplete trace: mark **60%** of the memory pairs as may-alias.
    • Over-approximation.

Incomplete trace
Why Deep Learning

• Previous applications demonstrated that DL can be used to learn patterns from input and assign each input an label.

• For alias analysis:
  • Input data (instructions): a sequence of machine code.
  • Labels: memory regions each instruction is attached.
  • E.g. `push 0x68732f2f`: data `[68 2f 2f 73 68]`.
    Label [stack stack stack stack stack].
Why Deep Learning

- Learning from the previous sequence of inputs adequately reflects the process of determining memory region access.

- Memory region that an instruction accesses can be determined:
  - Semantics of that instruction.
    - `push eax`: indicate a stack access.
  - Context indicated by previous instruction.

```
0:  8d 1c 24  lea   ebx,[esp]
3:  89 0b  mov   DWORD PTR [ebx],ecx
```

[ebx] indicates a stack access.
Challenges for DL

• Capture the sequential dependence within input sequence.
  • Each hex in the binary code sequence are highly correlated with each other.
• The bi-directional dependence.
  • Forward analysis procedure.
  • Backward analysis procedure.

00015670 <malloc>:
  15670: 53           push ebx
  ...
  1568a: 89 44 24 04  mov  DWORD PTR [esp+0x4],eax
  1568e: e8 6d b1 fe ff  call 800 <_libc_memalign@plt>
  15693: 83 c4 18     add  esp,0x18
  15696: 5b           pop  ebx
  15697: c3           ret

Indicates [eax] is a heap access.
Challenges for DL

• Catch the dependence between not only the dependency between and within instructions (input sequence) but also dependencies between adjacent labels.
  • Hexes in the same instruction have the same label.

push [esp]  Machine Codes  0xff 0x34 0x24  labels  stack stack stack
Recurrent neural network – dependence within Hexes

- Recurrent neural network.
  - Take sequential data as input.
  - Self-connected hidden units.
  - Model the dynamic temporal behavior for a time sequence.
- Types of hidden units:
  - SimpleRNN.
  - GRU.
  - LSTM (Capture long term dependence).
Teacher force – dependence between labels

• Integrate the label of late time step into the current input.
  • Remove the conditional independent assumption between adjacent labels.
Bi-directional RNN – backward dependence

• Bi-directional RNN - combines a RNN that moves forward, beginning from the start of the byte sequence, with another RNN that moves backward.
DEEPVSA: Deep learning Facilitated VSA

- Novel network structure: Bi-directional conditional LSTM.
  - Integrate LSTM, teacher force and Bi-directional connection together.
- Extension of conventional VSA.
  - Take the region predictions of DL model to determine non-alias relationships that the conventional VSA originally fails to identify.
Evaluation

• Experimental setup.
  • Dataset:
    • Randomly select 54 vulnerability reports from the Exploit Database Archive.
    • Run the PoC programs tied to corresponding vulnerabilities, trigger software failures and collect the 54 execution traces pertaining to the crashes.
  • Answer the following questions:
    • Does the bi-directional conditional LSTM exhibit better performance than other RNN architecture?
    • Can the memory regions identified improve the memory alias analysis detection performance?
Question one

• Comparison between different RNN structures.
  • Label the bytes tied to the execution traces with the memory regions.
  • Randomly divided the traces into 5 disjoint groups and train 5 distinct models for each network architectures.
    • Each model: take One group of traces as testing data and utilize the remaining to train our neural network models.

• Our model established highest precision and recall.

<table>
<thead>
<tr>
<th></th>
<th>Global</th>
<th>Heap</th>
<th>Stack</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bi-RNN</td>
<td>99.55%</td>
<td>99.33%</td>
<td>99.96%</td>
<td>99.75%</td>
</tr>
<tr>
<td>Bi-GRU</td>
<td>99.55%</td>
<td>99.49%</td>
<td>99.98%</td>
<td>99.80%</td>
</tr>
<tr>
<td>Bi-LSTM</td>
<td>99.55%</td>
<td>99.30%</td>
<td>99.97%</td>
<td>99.76%</td>
</tr>
<tr>
<td><strong>Our model</strong></td>
<td><strong>99.99%</strong></td>
<td><strong>99.79%</strong></td>
<td><strong>99.99%</strong></td>
<td><strong>99.88%</strong></td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Recall</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bi-RNN</td>
<td>99.50%</td>
<td>99.47%</td>
<td>99.94%</td>
<td>99.81%</td>
</tr>
<tr>
<td>Bi-GRU</td>
<td>99.54%</td>
<td>99.49%</td>
<td>99.94%</td>
<td>99.78%</td>
</tr>
<tr>
<td>Bi-LSTM</td>
<td>99.51%</td>
<td>99.55%</td>
<td>99.94%</td>
<td>99.81%</td>
</tr>
<tr>
<td><strong>Our model</strong></td>
<td><strong>99.88%</strong></td>
<td><strong>99.76%</strong></td>
<td><strong>99.97%</strong></td>
<td><strong>99.90%</strong></td>
</tr>
</tbody>
</table>
Question two

• Memory alias pairs detection.
• Apply the DL model to predict the memory regions tied to each trace.
• Pass the predictions to DEEPVSA and compute the non-alias pairs.
• Use conventional VSA as baseline for comparison.
Question two

- Memory alias pairs detection.
  - DEEPVSA improves the detection rate from 24.84% to 66.43%.
  - DEEPVSA does not introduce errors.

<table>
<thead>
<tr>
<th>Program</th>
<th>CVE/EDB-ID</th>
<th>LoC</th>
<th>TR?</th>
<th>VSA</th>
<th>DEEPVSA</th>
<th>Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Non-alias</td>
<td>Error rate</td>
<td>Non-alias</td>
</tr>
<tr>
<td>GnuPG-1.9.14</td>
<td>CVE-2006-3082</td>
<td>99053</td>
<td>✓</td>
<td>32.20%</td>
<td>0%</td>
<td>83.52%</td>
</tr>
<tr>
<td>podofo-0.94</td>
<td>CVE-2017-5854</td>
<td>60147</td>
<td>✓</td>
<td>0.49%</td>
<td>0%</td>
<td>27.86%</td>
</tr>
<tr>
<td>Average</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>24.84%</td>
<td>0%</td>
<td>66.43%</td>
</tr>
</tbody>
</table>
Summary

• DEEPVSA implements a novel RNN architecture customized for VSA.
  • Bi-directional conditional LSTM.
• DEEPVSA outperforms the off-the-shelf recurrent network architecture in terms of memory region identification.
• DEEPVSA significantly improves the VSA with respect to its capability in analyzing memory alias.
• DEEPVSA will enhance the accuracy and efficiency of the postmortem program analysis.
Thank you very much!