A Code Pirate’s Cutlass:
Recovering Software Architecture from Embedded Binaries

evm
@evm_sec
Motivation

• Problem space: vulnerability analysis for embedded devices, esp. real-time/embedded operating systems

• Goal: Expand previous work in call graph visualization for RE into automated call graph segmentation
  - “Bubble Struggle” by Marion Marschalek, RECON 2017
  - “Reverse Engineering with Hypervisors” by Danny Quist, RECON 2010

Understanding software architecture is critical to effective and timely vulnerability analysis in the embedded environment
RTOS / Embedded OS From An RE Perspective

• Single (often large) fully linked program
• One address space
• No clear distinction between
  - Application threads
  - Libraries
  - Operating system
• Usually distributed to licensees as source or object files
• No symbols (usually)
• Scattered debug prints (often)
• There are a gazillion of them
Towards Automated RE

- Objects / Libraries
- Subroutines / Functions
- Statements / Constructs
- Assembly / Opcodes

- Reverse engineers operate on at least 4 levels
- Usually when a new project gets started we are spinning our wheels a bit at the bottom in order to label enough functions to start to make sense of the bigger picture
- For ML/DL approaches – we are going to need methods to chunk up a large binary – and give a sense of context for each function
The CodeCut Problem

• Assumptions:
  - Embedded developers organize code into multiple source files
  - Source files are compiled into object files
  - Linker produces final binary that is a linear concatenation of object files
  - No intentional obfuscation
The CodeCut Problem

• Problem Statement: Given only call graph information for a large binary, recover the boundaries of the original object files

• Notes:
  - Essentially architecture independent (as long as a call graph can be generated through disassembly)
  - Inherent ambiguity: CodeCut algorithms might locate multiple functional clusters within an original source file - or combine two files because they are highly related

```
main.c  math_lib.c  net_lib.c  crypt_lib.c  std_lib.c
Compile

main.o  math_lib.o  net_lib.o  crypt_lib.o  std_lib.o
Link

main.o  math_lib.o  net_lib.o  crypt_lib.o  std_lib.o
CodeCut

Binary Program

main.o  unk_mod1.o  net_lib.o  unk_mod2.o  std_lib.o
```
Local Function Affinity Concept

#include <stdio.h>

int helper_1() {
    return helper_2()/100;
}

int helper_2() {
    ...
}

int more_complex() {
    ...
    while (helper_1() < 100) {
        foo = helper_2() % 20;
    }
    ...
}

void main_functionality() {
    more_complex();
    ...
    while (helper_2() > 1000) {
        foo = helper_1();
        bar = more_complex();
    }
}
Local Function Affinity Concept

- If we eliminate external calls…
Local Function Affinity Concept

```c
#include <stdio.h>
int helper_1() {
    return helper_2() / 100;
}
int helper_2() {
    ... 
}
int more_complex() {
    ...
    while (helper_1() < 100) {
        foo = helper_2() % 20;
    }
    ...
}
void main_functionality() {
    more_complex();
    ...
    while (helper_2() > 1000) {
        foo = helper_1();
        bar = more_complex();
    }
}
```

- If we eliminate external calls…
- Directionality of calls at the beginning of the module is in the positive direction
Local Function Affinity Concept

```c
#include <stdio.h>
int helper_1() {
    return helper_2() / 100;
}
int helper_2() {
    ...
}
int more_complex() {
    ...
    while (helper_1() < 100) {
        foo = helper_2() % 20;
    }
    ...
}
void main_functionality() {
    more_complex();
    ...
    while (helper_2() > 1000) {
        foo = helper_1();
        bar = more_complex();
    }
}
```

- If we eliminate external calls…
- Directionality of calls at the beginning of the module is in the positive direction
- Directionality of calls generally switch to the negative direction towards the end of the module
Local Function Affinity Concept

- If we eliminate external calls…

- Directionality of calls at the beginning of the module is in the positive direction

- Directionality of calls generally switch to the negative direction towards the end of the module

- We can detect edges by finding the switch from negative back to positive
Local Function Affinity Concept

- If we eliminate external calls...

- Directionality of calls at the beginning of the module is in the positive direction

- Directionality of calls generally switch to the negative direction towards the end of the module

- We can detect edges by finding the switch from negative back to positive
Local Function Affinity Definition

\[ Affinity(f) = \frac{\sum_{x \in \text{references}(f), \text{sign}(x - f) * \log(|x - f|)}}{|\text{references}(f)|} \]

Where \( \text{references}(f) \) is defined as the set of functions that call \( f \) or are called by \( f \) for which the distance from \( f \) to the function is below a chosen threshold. Multiple references are counted.

- Using fixed threshold of 4K*
- Edge Detection*:
  - General negative trend
  - Change to positive value (\( \Delta > 2 \))
  - Treat calls to / calls from as separate scores – for functions without one of the scores, interpolate from last score

* room for improvement!
Call Directionality Metric
Module-to-Module Call Graph (Auto-Generated)
CodeCut Success Metric

<table>
<thead>
<tr>
<th>Code</th>
<th>.map file (ground truth)</th>
<th>.map file (alg output)</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>3b 7f 00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 ff 11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 00 d1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3c 01 00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a7 12 01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 2d e9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f0 41 76</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4b 14 46</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>76 4a 7b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44 9b 58</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0e 46 19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>68 a6 b0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25 91 05</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>46 98 46</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>08 b1 06</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b1 bc b9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>71 4a 72</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4b 7a 44</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7b 44 92</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20 00 92</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 93 02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90 03 21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63 20 05</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aa 80 23</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cc f7 e8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ed 30 b1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6c 4b 63</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20 03 21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>05 aa 7b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44 cc f7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e6 ed 00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20 bc e0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 f5 1c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42 92 f8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>61 30 01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2b 40 f0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>98 80 00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f5 1f 47</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>07 f1 78</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 ff f7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>93 fe c8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b9 30 70</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2018 The Johns Hopkins University Applied Physics Laboratory LLC. All Rights Reserved.
## LFA Results to Date

<table>
<thead>
<tr>
<th>System</th>
<th>Match (%)</th>
<th>Gap (%)</th>
<th>Underlap (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gnuchess (x86)</td>
<td>76.1</td>
<td>3.2</td>
<td>20.7</td>
</tr>
<tr>
<td>PX4 Firmware/NuttX (ARM)</td>
<td>82.2</td>
<td>13.6</td>
<td>4.2</td>
</tr>
<tr>
<td>GoodFET 41 Firmware (msp430)</td>
<td>76.1</td>
<td>0</td>
<td>23.9</td>
</tr>
<tr>
<td>Tmote Sky Firmware/Contiki (msp430)</td>
<td>93.3</td>
<td>0</td>
<td>6.7</td>
</tr>
<tr>
<td>NXP Httpd Demo/FreeRTOS (ARM)</td>
<td>86.7</td>
<td>1.4</td>
<td>11.9</td>
</tr>
</tbody>
</table>
Future Work

• Combine LFA with graph algorithm solutions to CodeCut

• Include global data references

• LFA improvements:
  - Basic similarity score metric for functions with no score (eliminate “gaps”)
  - Dynamically adjust “external” threshold in LFA score (currently fixed)
  - Experiment with more advanced edge detection
  - Possibly combine threshold and edge detection experiment
A Code Pirate’s Cutlass:
Recovering Software Architecture from Embedded Binaries

evm
@evm_sec