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

evm
@evm_sec
Motivation

• Much of infosec is built on top of reverse engineering (RE)

• RE is manually intensive and requires multi-domain expertise, particularly for embedded systems

• Embedded systems
  - Combine OS, libraries, and application code into a single program space
  - Binary is fully linked with no symbols (usually)

• Previous research in RE has focused on
  - Code-to-code translation: Binary -> Intermediate Language -> High Level Language
  - Function-level matching
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
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

- 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

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

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)
LFA Results to Date

<table>
<thead>
<tr>
<th></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>
A Maximum Cut Graph Algorithm

- \( \text{Weight}(C) = \frac{\sum_{E \in \text{crossings}(C)} |E|}{|\text{crossings}(C)|} \)

where \( \text{crossings}(C) \) is defined as the set of edges (calls) that “cross” the cut address

- Algorithm:
  - For every possible cut \( C \), calculate \( \text{Weight}(C) \) and choose \( C \) with maximum weight
  - Remove edges that cross \( C \) from graph
  - Divide graph into two subgraphs
  - Recursively evaluate subgraphs, stop when modules are below a chosen threshold
Show Me The Code!

CodeCut is available at:

http://github.com/jhuapl/CodeCut

(LFA only for now)

Contact Info:

@evm_sec
evm.ftw@gmail.com
A Code Pirate’s Cutlass:
Recovering Software Architecture from Embedded Binaries

evmev
@evm_sec