JOHNS HOPKINS
APPLIED PHYSICS LABORATORY

A Code Pirate’s Cutlass:

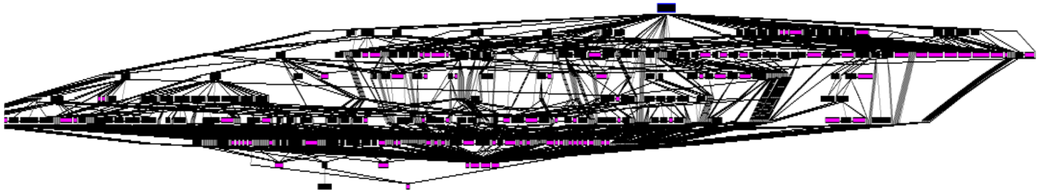
Recovering Software Architecture from Embedded Binaries

evm
@evm_sec

Motivation

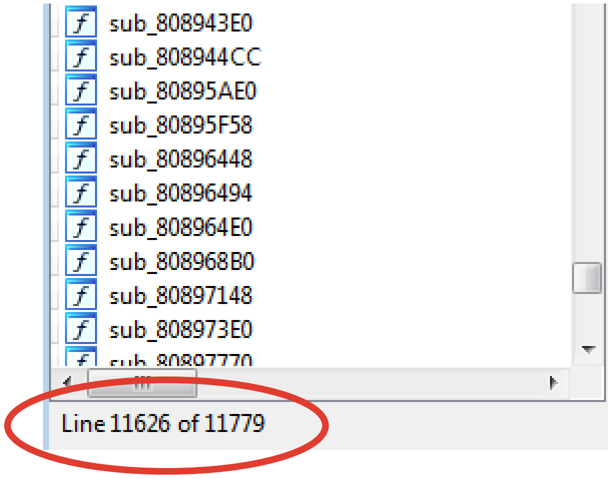
- Problem space: vulnerability analysis for embedded devices, esp. real-time/embedded operating systems

- 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

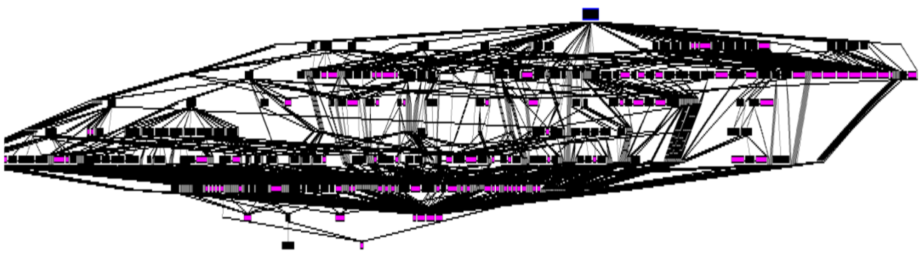


Understanding software architecture is critical to effective and timely vulnerability analysis in the embedded environment

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



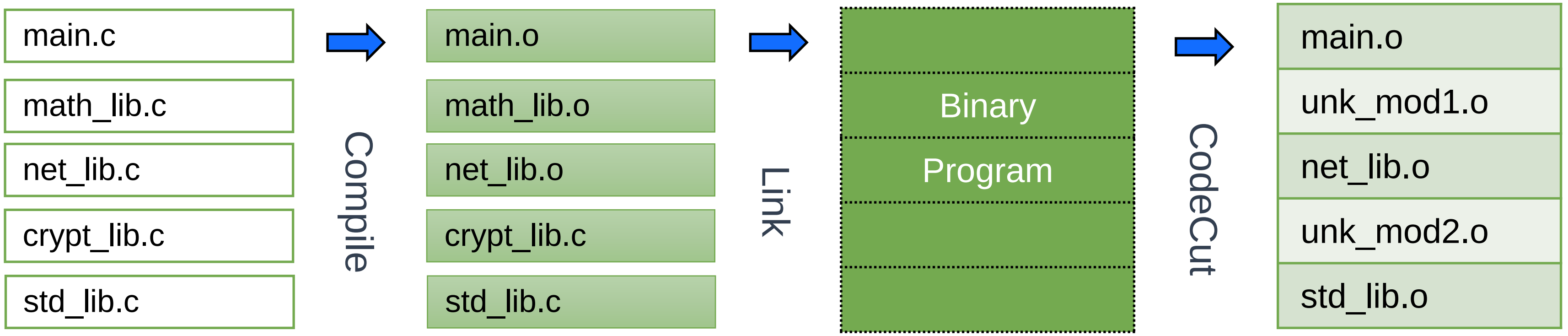
Towards Automated RE

- 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

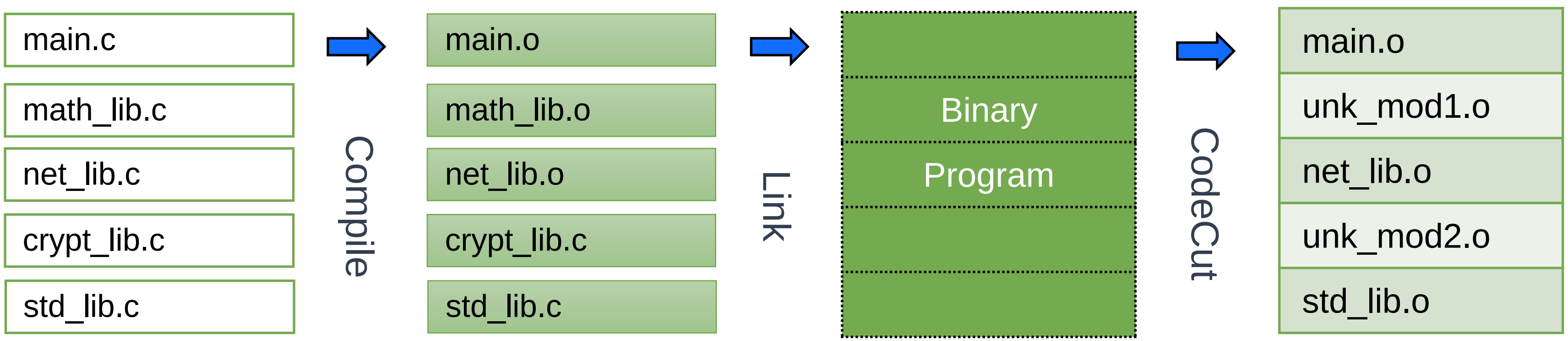
- Assumptions:
  - Embedded developers organize code into multiple source files
  - Source files are compiled into object files
  - Linker produces final binary that is a linear concatenation of object files
  - No intentional obfuscation



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



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

```
#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…

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();
    }
}
```

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

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();
    }
}
```

- 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

```
#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
- We can detect edges by finding the switch from negative back to positive

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();
      }
}
```

- 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

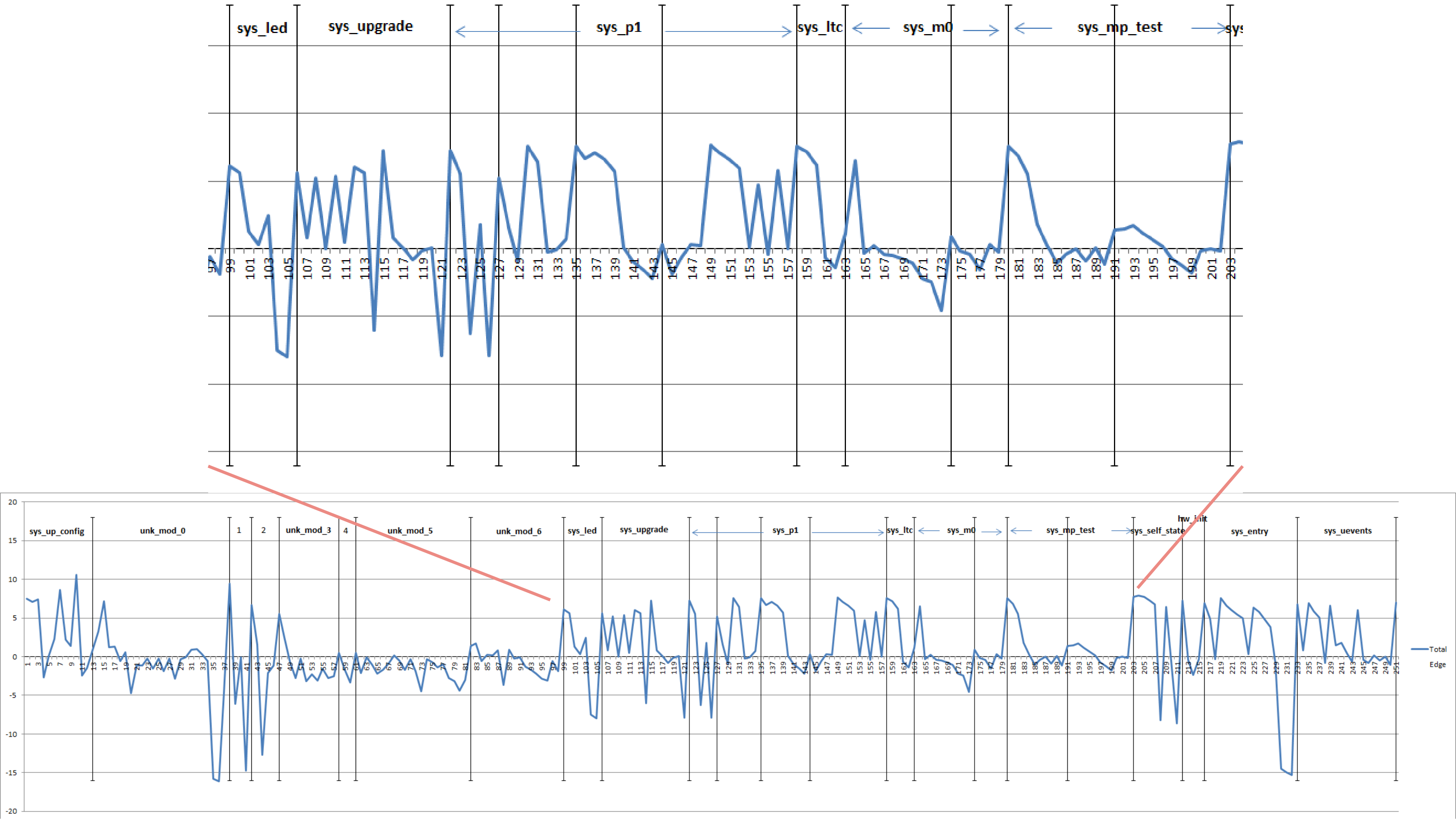
$$Affinity(f) = \frac{\sum_{x\, \in references(f),} sign(x - f) * Log(|x - f|)}{|references(f)|}$$

Where *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.

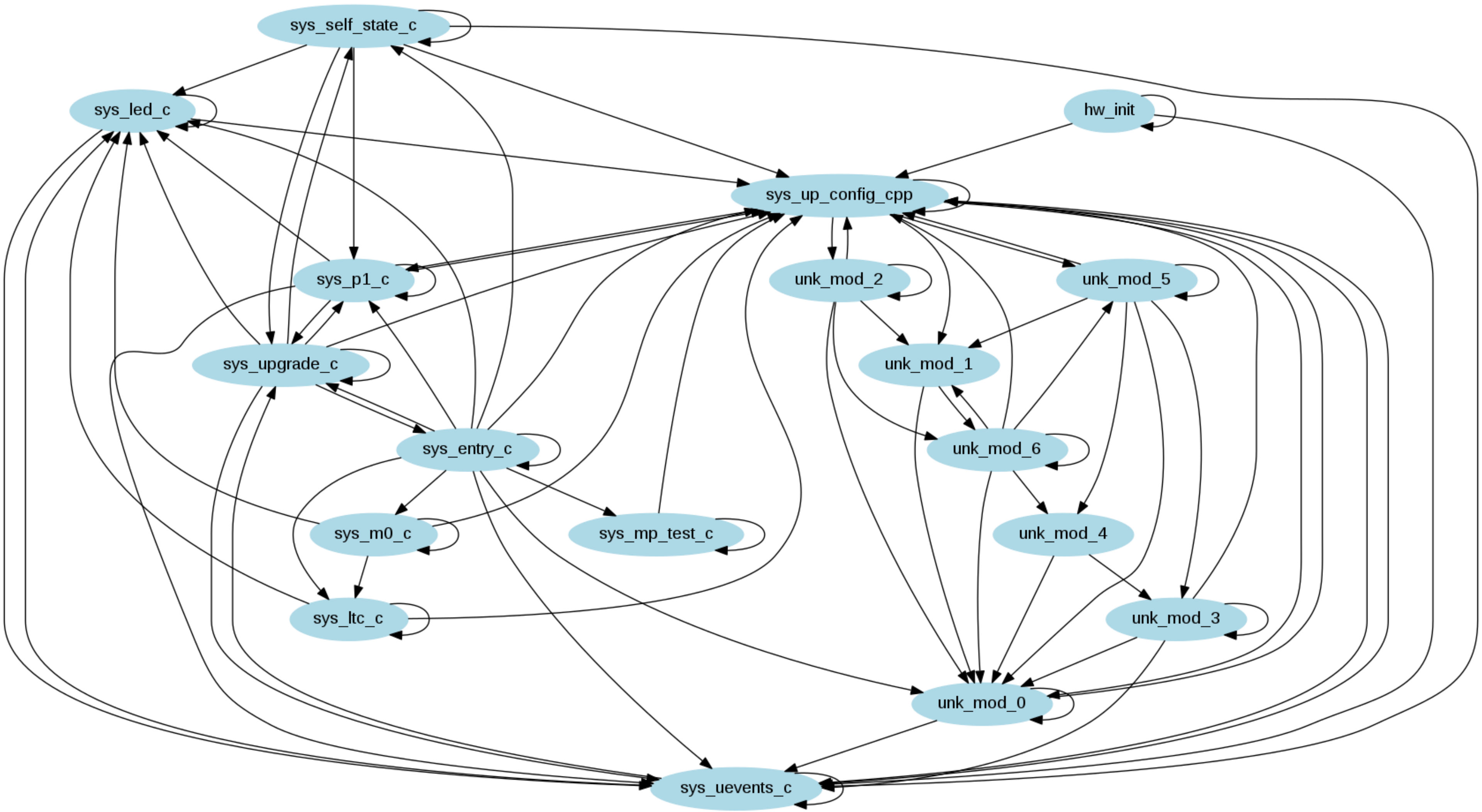
- Using fixed threshold of 4K*
- Edge Detection*:
  - General negative trend
  - Change to positive value ($\Delta > 2$)
  - Treat calls to / calls from as separate scores – for functions without one of the scores, interpolate from last score

* room for improvement!

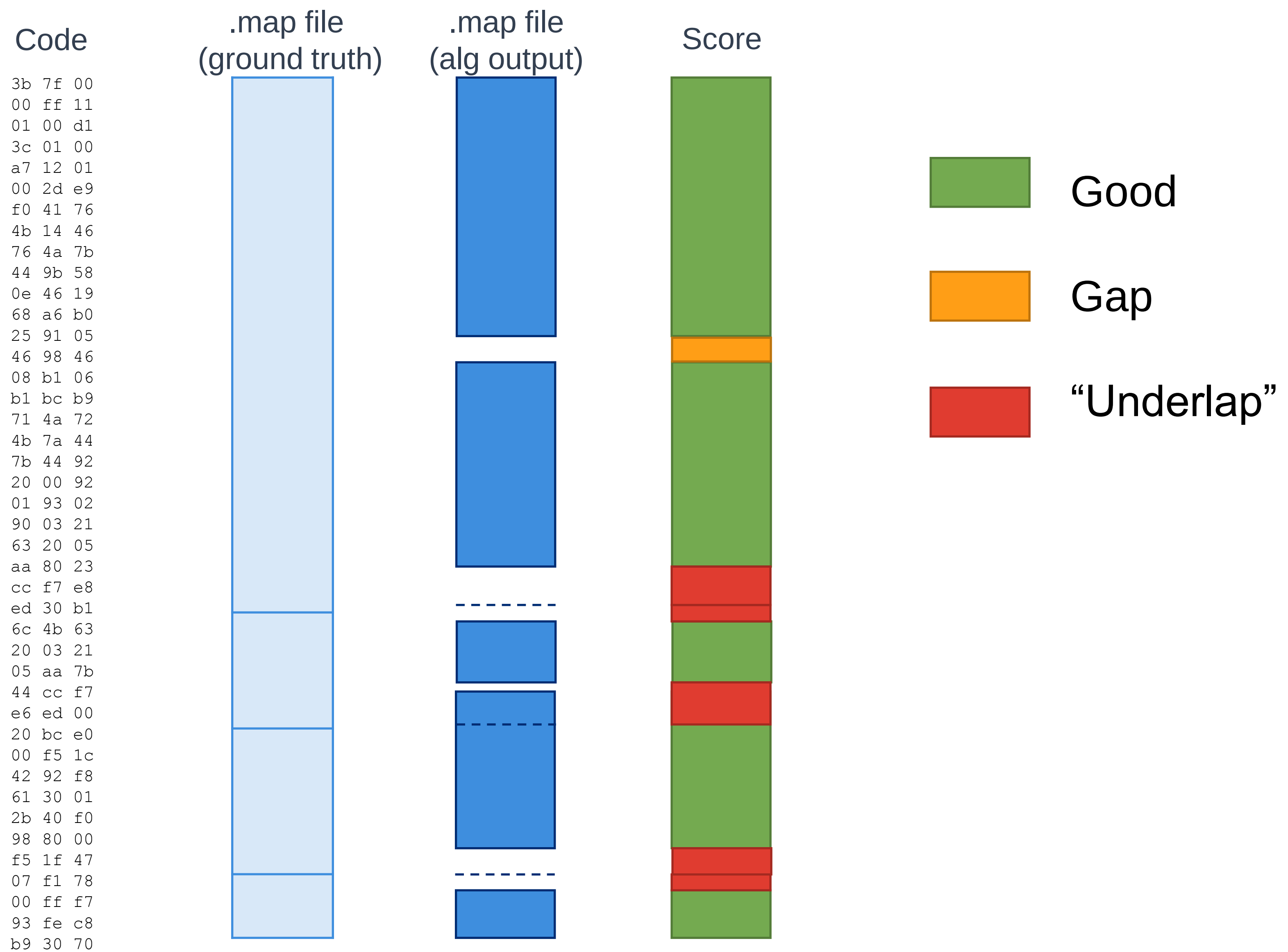
Call Directionality Metric



Module-to-Module Call Graph (Auto-Generated)



CodeCut Success Metric



LFA Results to Date

	Match / Gap / Underlap (%)		
• Gnuchess (x86)	76.1	3.2	20.7
• PX4 Firmware/NuttX (ARM)	82.2	13.6	4.2
• GoodFET 41 Firmware (msp430)	76.1	0	23.9
• Tmote Sky Firmware/Contiki (msp430)	93.3	0	6.7
• NXP Httpd Demo/FreeRTOS (ARM)	86.7	1.4	11.9

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

JOHNS HOPKINS
APPLIED PHYSICS LABORATORY

A Code Pirate’s Cutlass:

Recovering Software Architecture from Embedded Binaries

evm
@evm_sec