

## 20:10 A Code Pirate's Cutlass: Recovering Software Architecture from Embedded Binaries

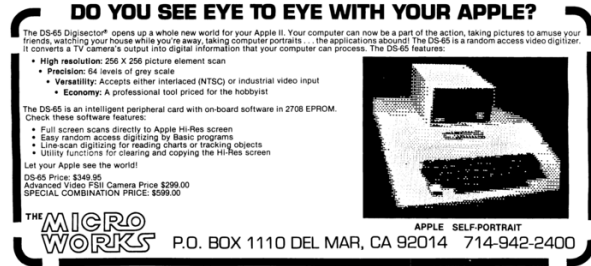
by *evm*

He looks around, around  
He sees angels in the architecture  
Spinning in infinity  
He says Amen! and Hallelujah!  
- Paul Simon, "You Can Call Me Al"  
(which was probably not written  
about software RE)

Software RE underlies much of the work in the cyber landscape for both defensive and offensive operations.

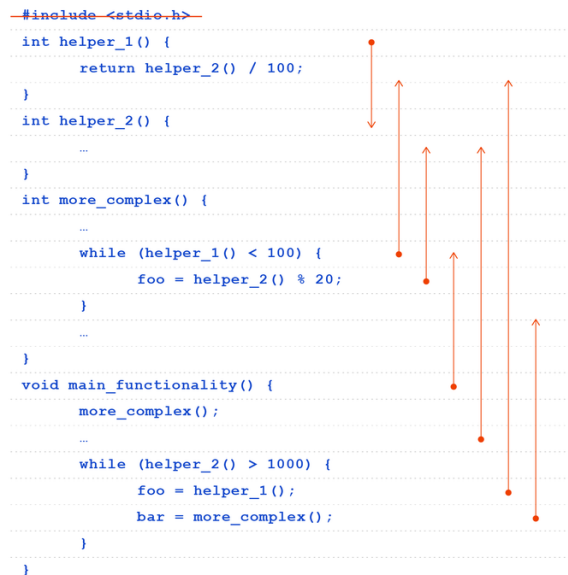
When developing complex programs, it is common to segment functionality of code into multiple source files. These source files are compiled into multiple object files and then linked into an executable program. The object files contain pieces of information (such as the developer-given names of functions and global data structures) that the linker uses to determine relationships between them. Once the linker produces the final executable, all the intermediate developer-generated information is gone (unless for some reason debugging information is included, which rarely happens in production code). See Figure 1 for an illustration of this process.

This means that software reverse engineers approaching a new target are usually dealing with a fully linked binary with no symbols included. However, we know that the binary is just a conglomeration of the original object files, usually in the exact order they were passed to the linker. Usually software reverse engineers are interested in a specific cross section of the binary associated with either a particular high-level function ("how does this program handle network authentication?") or whether vulnerable points in the code can be reached from a particular entry point. Often software reverse engineers use different clues to find either the functionality they are interested in or the areas they think might be vulnerable. Eventually after many hours of the analyst's time, the structure and design of the code may become apparent. What if the structure and design of code could be extracted in an automated way? How much faster and more effective could we make RE if we were able to work from the beginning by analyzing the design of the program instead of starting from a sea of subroutines?



### Defining the Metric

The concept is pretty simple. Local function affinity (LFA) is like a force vector, showing which direction a subroutine is pulled toward based on its relationship to nearby subroutines. Consider your average C source code file - and ignore external function calls for the moment. As you move from the beginning of the file down to the bottom, calls start in the positive direction (down) and eventually switch to the negative direction (up). The idea is that when we look at the binary, we should be able to detect the switch from the negative direction back to positive at the beginning of the next object file.



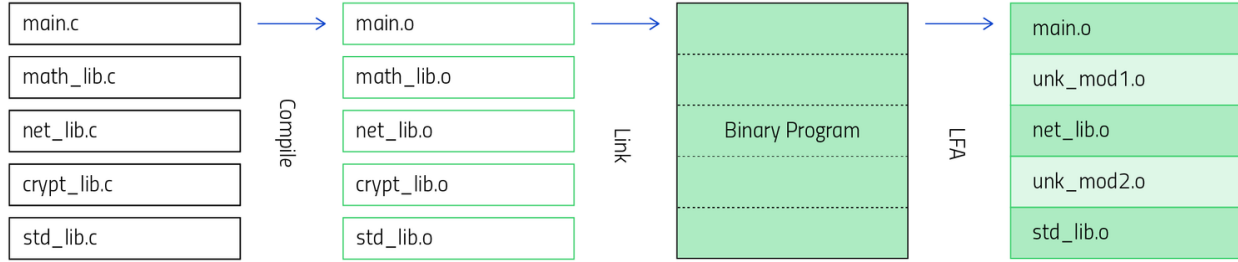


Figure 1. Illustration of compilation, linking, and what this research is attempting to produce. Note: This is greatly oversimplified (e.g., the standard library often consists of hundreds of object files).

So how do we deal with external calls? For now, LFA just discards any function calls over a fixed threshold, which currently has been set at 4 KB. Admittedly this isn't a great way to do it, and later I'll talk about some ways this might be improved.

We need to combine both outgoing function references (calls FROM this function to other functions) and incoming function references (calls TO this function from other functions) to include helper functions that don't make calls. Even with the external calls "eliminated," we want to weight our metric toward nearby neighbors. So we define the metric this way:

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

where  $neighbors(f)$  is defined as the set of functions (i.e., their address in the memory map) 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.

For practical purposes, in my current implementation of LFA, I treat the outgoing and incoming references as separate scores, and if either is zero, I interpolate a new score based on the previous score. This helps to smooth out the data.

## Detecting Object Boundaries

For now, LFA has a simple edge-detection metric, which is simply a change from negative values (two of three previous values are negative) to a positive value where the difference is greater than 2. During initial research, a colleague suggested a simple metric like this due to the irregularity of the signal (i.e., due to the varying sizes of object files). This edge-detection strategy can most certainly be improved upon (which will be discussed later).

I should also note here that when a function has

no LFA score (meaning it either has no references, or all references are above the external threshold), my current implementation treats it like it isn't there. This creates gaps between object files.

## Extracting Software Architecture

Once approximate object file boundaries are extracted, we can produce a software architecture picture by generating a directed graph where each object is a node, and edges between nodes represent calls from any function in the first object to any function in the second object.

With the object file boundaries approximately identified, we can also make use of debugging string information in the binary. The current LFA implementation looks at possible source file names as well as common words, bigrams and trigrams in order to guess a possible name for the object.

Figure 2 shows an example software architecture diagram automatically extracted from a target binary using LFA. Some interesting features are readily apparent in this graph, which are not readily discernible by other means. It is readily apparent which objects are most commonly referenced in the target program (e.g. `sys_up_config` and `unk_mod_5`). Notice also how unknown modules 1-6 form a sub-graph that is only reachable from `sys_up_config`. This indicates that these objects are only used by `sys_up_config` and not directly called by any other object. This means they are essentially a library dependency for `sys_up_config` and can be safely ignored by the RE analyst (unless the functionality of `sys_up_config` is of interest).

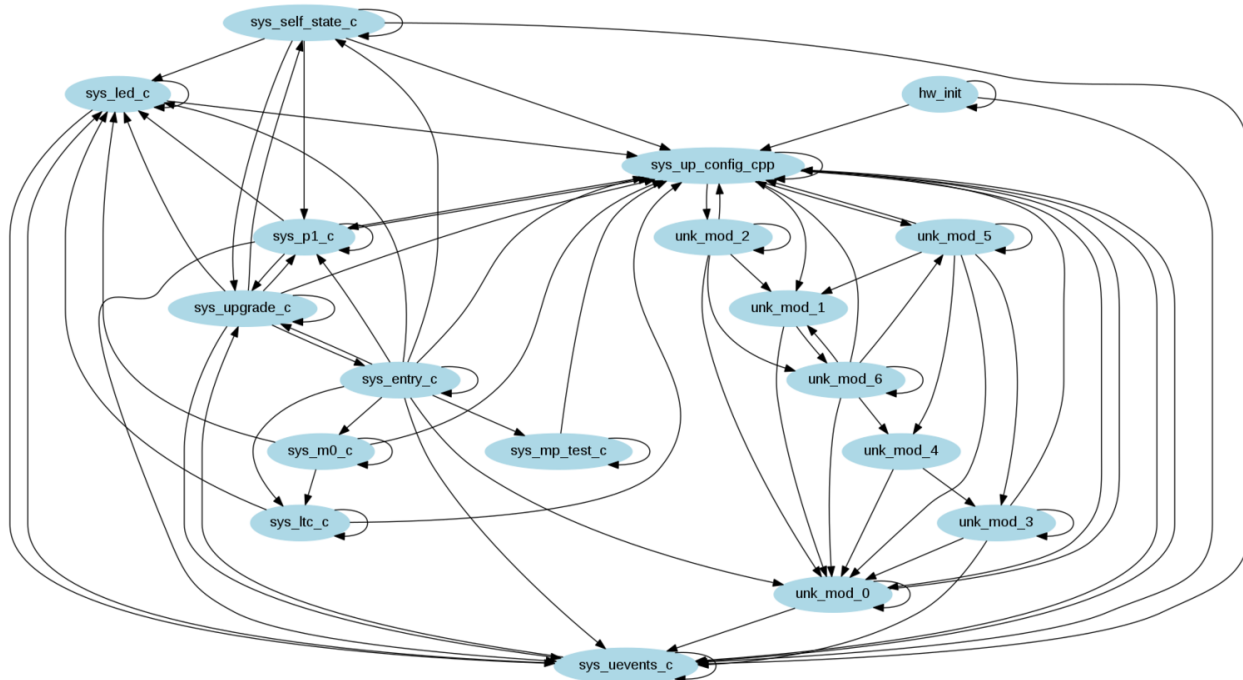


Figure 2. Automated software architecture graph produced by LFA, with objects/modules named by source file string references.

## Measuring Success

As far as I can tell (and dear reader, I would humbly welcome your education on this subject if you have further information), measuring success in solving this problem is somewhat unusual and difficult for a couple of reasons. We want to credit the algorithm with success when it identifies smaller groups of functionality within an original source file. For instance, if a very large source file contains three groups of related functions, we want to give the algorithm credit if it identifies these three groups as separate objects. We also want to give credit when the algorithm defines two adjacent, closely related objects as a single thing.

LFA outputs a .map file, which is compared against the .map file produced by the compiler during the build (the ground truth). First we define a process of reconciliation, where we combine modules (objects) in the ground truth file and in the algorithm’s .map file, to produce the best alignment possible between the maps. To do this we start with the first module in both maps. We combine whichever module is shorter with subsequent modules in that map to produce the best alignment with the module from the other map. During this pro-

cess, whenever there are gaps between modules in the algorithm’s list, we add these to the “gap area” count. We assume that the ground truth .map file is contiguous.

Once the maps are reconciled, for each module in the algorithm’s map, we score the area that matches the ground truth map and also score the “underlap” (areas of the ground truth module not covered by the algorithm’s module). The final score is then a combined result of match, gap, and underlap percentages for the binary. A perfect score would be a 100% match, with no gaps or underlaps. See Table 3 for a list of results to date.

**CATALOGUE**



# FREE

Now is the time to buy a **PIANO OR ORGAN** from the largest manufacturer in the world, who sell their instruments direct to the public at wholesale factory prices. Don't pay a profit to agents and middlemen. **TERMS** to suit all. No money asked in advance. Privilege of testing organ or piano in your own home 30 days. No expense to you if not satisfactory. Warranted 25 years.

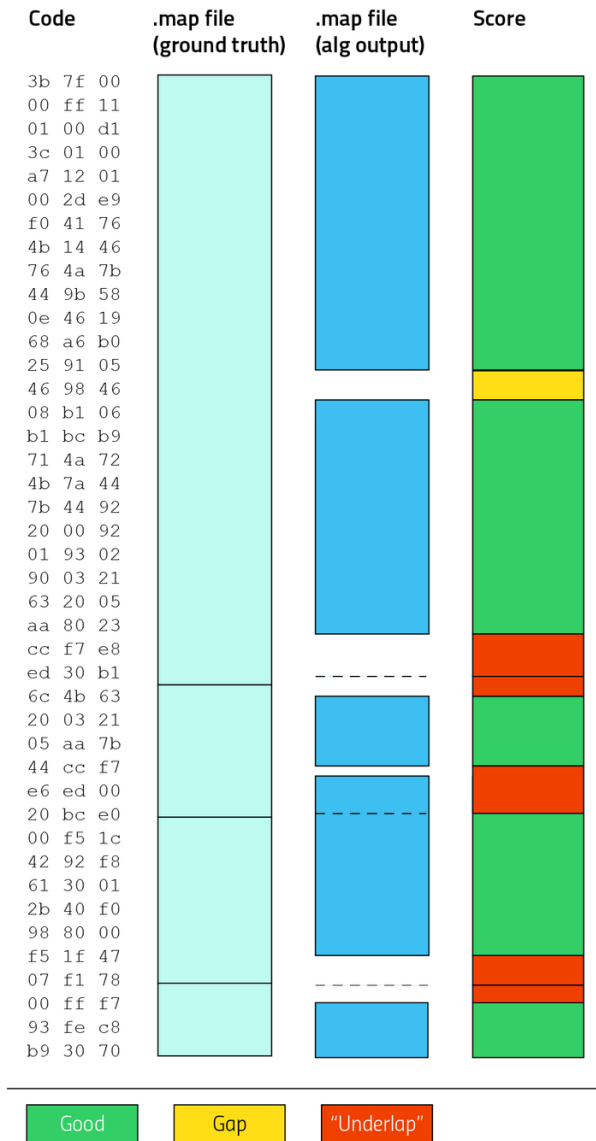
**REFERENCE** Bank references furnished on application: the editor of this paper; any business man of this town, and to the thousands using our instruments in their homes. A book of testimonials sent with every catalogue. As an advertisement we will sell the first Piano in a place for only **\$159**. The first Organ only **\$25**. **Stool, Book, etc. FREE**. If you want to buy for cash, **BUT DON'T BUY UNTIL YOU** **Write Us.** **BEETHOVEN PIANO & ORGAN CO.,** P. O. Box 882 WASHINGTON, N. J.



BUT DON'T BUY UNTIL YOU Write Us. BEETHOVEN PIANO & ORGAN CO., P. O. Box 882 WASHINGTON, N. J.

Name/operating system (architecture)	Match, %	Gap, %	Underlap, %
Gnuchess (x86)	76.1	3.2	20.7
PX4 Firmware/NuttX (ARM)	82.2	13.6	4.2
GoodFET41 Firmware (msp430)	76.1	0.0	23.9
Tmote Sky Firmware/Contiki (msp430)	93.3	0.0	6.7
NXP HTTPD Demo/FreeRTOS (ARM)	86.7	1.4	11.9

Figure 3. LFA results to date. The algorithm has a high gap score on the PX4 firmware due to a few very large functions that generate no LFA score.



## A Max Cut Graph-Based Algorithm

Many graph algorithms that deal with segmentation are encumbered by the fact that nodes exist in two or three dimensions, meaning that there are factorial possibilities for “cuts” in the graph. Not so for a binary. Although the graph representation may be complicated, a binary is a one-dimensional structure, a number line. Using this to my advantage I developed an algorithm which segments the binary by cutting it into two pieces, then recursively cutting those pieces until a threshold is reached. In the binary the possible “cuts” are between the end of one function and the beginning of the next (one possible cut for every function in the binary). These possible cuts are scored by scoring the average of the call distances for all calls that metaphorically “pass over” the cut address. The higher the average call score, the less likely the two functions on either side of the cut are to be part of the same object (since short range inter-object calls would lower the score).

Pseudocode of the maximum cut object segmentation algorithm is shown in Figure 4.

The algorithm runs in  $O(n \log n)$  for speed, and  $O(n^2)$  for memory usage, although memory usage could be reduced if old copies of the graph could be freed. From limited evaluation, MaxCut seems to work at least as well as LFA in most cases, see results in Table 5.

SAM COUPE AND SPECTRUM MAGAZINE!  
 PROGRAMS, UTILITIES, INFO, IDEAS! NEWS, REVIEWS AND HOMEGROWN SOFTWARE MONTHLY SINCE 1987!  
**"OUTLET"** AND HELP PAGES, SERIOUS SOFTWARE  
 SPECIAL OFFER! Latest issue £2.50 to newcomers on:-  
 +3, DISCIPLE/+D, MICRODRIVE, OPUS, TAPE, SAM DISC  
 CHEZRON SOFTWARE, 605 LOUGHBOROUGH RD., BIRSTALL, LEICESTER LE4 4NJ

<sup>46</sup>Jin, Wesley, et al. “Recovering c++ objects from binaries using inter-procedural data-flow analysis.” Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014. ACM, 2014.

<sup>47</sup>Yoo, Kyungjin, and Rajeev Barua. “Recovery of Object Oriented Features from C++ Binaries.” APSEC (1). 2014.

```

function make_cut(start, end, graph):
2  for node in graph.nodes:
    cut_address = node.address - 1
4  weight[cut_address] = 0
    edge_count = 0
6  for edge in graph.edges:
    if edge crosses cut_address:
8      weight[cut_address] += edge.length
        edge_count +=1
10 if edge_count == 0:
    return cut_address
12 else:
    weight[cut_address] = weight[cut_address] / edge_count
14 return address with maximum weight

16 function do_cutting(start, end, graph):
    if (end - start > THRESHOLD) and graph.nodes > 1:
18     cut_address = make_cut(start, end, graph)
        do_cutting(start, cut_address, subgraph(graph, start, cut_address))
20     do_cutting(cut_address+1, end, subgraph(graph, cut_address+1, end))
    else:
22     print "Object boundary from " start " to " end

24 main:
    start = binary start address
26     end = binary end address
    graph = graph of binary (functions are nodes, calls are edges)
28     do_cutting(start, end, graph)

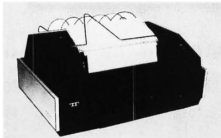

```

Figure 4. Pseudocode of the Maximum Cut Object Segmentation Algorithm

**TELETYPES®**

**IMMEDIATE DELIVERY**

MODEL 40 300 LPM PRINTERS

- Mechanism or complete assembly
- 80-column friction feed
- 80-column tractor feed
- 132-column tractor feed


**INTERFACES**

- EIA-RS232
- Simplified EIA-like interface
- Standard serial interface
- Parallel device interface

**FEDERAL Communications Corporation**  
11126 Shady Trail, Dallas, Texas 75229, (214) 620-0644,  
TELEX 732211 TWX 910-860-5529

**TRS 80**

**TRS-80 Model I**




Level II 16K, 26-1056

**MODEL 3**  
call today!

**689**

**\$3450**

**Model II**



64K, 1-Disk TRS-80 Model II System

We accept check, money order or phone orders with Visa or Master Charge. (Shipping costs added to charge orders.)

Computers Unlimited 419-332-4881 Collect

1528 DAK HARBOR BLVD. FLEMING, OHIO 43002

Name/operating system (architecture)	Match, %	Underlap, %
Gnuchess (x86)	92.8	7.2
PX4 Firmware/NuttX (ARM)	98.9	1.1
GoodFET41 Firmware (msp430)	97.0	3.0
Tmote Sky Firmware/Contiki (msp430)	89.6	10.4
NXP HTTPD Demo/FreeRTOS (ARM)	94.8	5.2

Figure 5. MaxCut results to date.

## Related Work

Much of the related work in this area involves locating objects or object boundaries in C++ code, using either static analysis,<sup>46 47</sup> or sometimes a combined static and dynamic analysis approach.<sup>48</sup> This work is purely based on static analysis and will work on C or C++ code, it does not use C++ features like run-time type information (RTTI). It makes use of the idea that linkers usually concatenate object files that they receive as input into the output binary.

Some work exists in generating design diagrams (e.g. UML) from source code.<sup>49 50</sup> This work shows generating design diagrams directly from binaries by first locating object file boundaries. It also presents a metric for measuring the effectiveness of future solutions to the problem of locating object file boundaries is presented.

**IMPORTANT NOTICE**

There are thought to be approximately 20 virus programs circulating in the Atari ST community worldwide

**Protect your ST with**

**THE VIRUS DESTRUCTION UTILITY 3.1**

**ONLY £6.95 INC P&P**

Excel Software are the sole U.K. Agents for the above product  
(Dealer enquiries welcome)

Excel Software also operate a large public domain software library with guaranteed virus free software!

*Send a 19p stamp or call us today for our latest catalogue*

**EXCEL SOFTWARE, PO BOX 159, STOCKPORT SK2 6HN**  
**TELEPHONE: 061-456 9587 (After 6pm)**

## Future Work

The possibilities for experimentation here are endless, and much of my motivation to publish this work is to get others to play around with LFA and Max Cut and brainstorm new possible ways to solve the problem. Thank you to everyone I have brainstormed ideas with.

First off, for LFA I am not convinced that taking the logarithm of distance is the best way to score. I believe using the inverse square of distance would be a little too drastic, but this could use some experimentation. An area for improvement is the “threshold” as a placeholder for removing external functions. A simple experiment might be to vary the threshold and run LFA on the data set, looking for the best result. Another area for improvement is edge detection. One possibility would be to generate the LFA curve for a variety of object files from data sets, and then generate a characteristic LFA curve. This characteristic curve could be convolved with the LFA signal or could be used with a dynamic threshold approach (i.e., the “external” threshold is varied until the signal best matches the characteristic curve).

For Max Cut, some development needs to happen to allow it to produce output matching LFA’s output, and then it can be tested on the current dataset.

I envision LFA/Max Cut as one day being a piece of a multilayered, deep learning system for translating binary code into natural language automated static reverse engineering. The LFA source code for this article is available attached to this PDF and through Github.<sup>51</sup>

<sup>48</sup>Tonella, Paolo, and Alessandra Potrich. “Static and dynamic C++ code analysis for the recovery of the object diagram.” ICSM. IEEE, 2002.

<sup>49</sup>Tonella, Paolo, and Alessandra Potrich. “Reverse engineering of the interaction diagrams from C++ code.” Software Maintenance, 2003. ICSM 2003. IEEE, 2003.

<sup>50</sup>Sutton, Andrew, and Jonathan I. Maletic. “Mappings for accurately reverse engineering UML class models from C++.” Reverse Engineering, 12th Working Conference on. IEEE, 2005.

<sup>51</sup>git clone <https://github.com/JHUAPL/CodeCut> || unzip pocorgtfo20.pdf CodeCut.zip