Reverse Engineering: Determining Algorithms in Binaries

George Brinzea
Dylan Fistrovic
Javid Habibi
Rahul Mohandas

Purdue University

Abstract

Three methods to identify algorithms in binary are presented. These algorithms can be used to match algorithms inside an binary to aid in the analysis of the executable’s instructions.

The most interesting applications in the realm of reverse engineering are in malware analysis where the analysis of a series of viruses to detect what algorithms are being utilized in a malicious fashion.

A framework implementation utilizing the described methods is presented, along with empirical data about it’s performance when analyzing multiple binaries and identifying different algorithms within each one.

Summary

We developed three separate plugins to solve the algorithm analysis problem. We developed two generic solutions, and one specific algorithm solution. The specific algorithm we detected was Bubble Sort. Our first generic solution was the Signature Vector approach, which created a function signature using instruction counts, ratio of code blocks to edges, and back edges in the current functions graph. Our second generic solution was using Small Primes Product to determine if the functions were functionally the same.

Our bubble sort solution yielded the best results with a 100% accuracy with all the binaries we tested. Our Signature Vector approach was not as successful, but it was able to return matches, however multiple false positives were present. Our Small Primes Product approach was successful, however fine-tuning of thresholds is necessary.

Solutions

We developed three separate algorithm analysis plugins. Specifically we developed one specific analysis algorithm solution, and two generic algorithm analysis solutions. Using the generic solutions in conjunction, these plugins can mitigate false positives.

Specific Algorithm Solution: Bubble Sort

For our specific algorithm solution, we concluded that detecting Bubble Sort would be optimal. This is due to the fact that Bubble Sort can be separated into four fundamental parts, additionally this algorithm is very similar to insertion sort, which we used for testing. The four fundamental parts of Bubble Sort are:

- Two loops, we must be able to detect the inner and outer loop
- A comparison statement
- A swap

Part 1: Loop Detection.

Our method for detecting loops began by creating a custom graph class to supplement the graph structure define by IDA. For each function defined in the executable, we created a new graph object to store the execution path. We then performed a depth first search to construct a list of backedges in the graph. Using these backedges we were able to determine if a basic block is inside a loop by checking if this basic block is dominated by the starting loop’s basic block.

Part 2: Swap and Comparison detection.

This step was accomplished by iterating over the function's assembly instructions. As we progressed through the function's instructions we kept track of the current registers values, and store them in a Python dictionary. However, we only track these registers: eax, ebx, ecx, edx, edi, esi. We do not track the esp, and ebp registers because all calculations are done in reference to the current stack frame. In addition, to tracking the registers for each basic block we create a graph structure of the block's mov instructions.

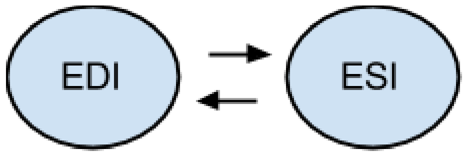


Figure: 1a

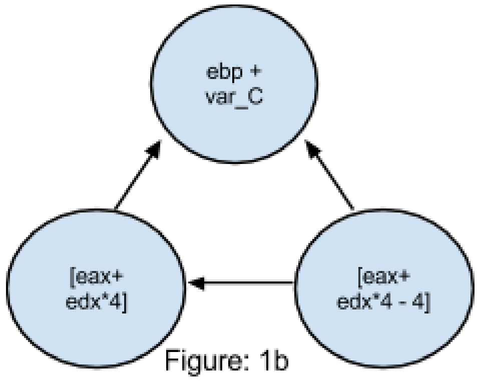


Figure: 1b

Using the graph structure we then determine if there is a cycle from two memory locations specified in the basic blocks last mov instruction. When checking for the cycle, if the memory location is a register we replace the register's current value with the value from our Python dictionary.

For example, in Figure 1a, this is a graph structure that can be created from a basic block only containing two mov instructions. This graph structure is generally formed when compiler optimizations are turned on for both Visual Studio and g++. In Figure 1b, a temporary variable is present, this graph structure is created when compiler optimizations are disabled. Both of these figures depict a swap.

Once a swap is found, we then check the last compare instruction to determine if the variables being compared are the same as the ones being swapped. If the variable being compared are the same as the ones being swapped in the current block, we then check if the current block is nested inside two loops by following the steps explained in Part1: Loop Detection. If the block is inside a set of nested loop we can safely conclude that the function contains Bubble Sort.

Generic Algorithm Analysis Solution

One of the goals we wanted to achieve for this project was to develop a solution that could work for any arbitrary algorithm in any arbitrary binary. While developing a custom fit approach for a specific algorithm will more accurately and definitively be able to detect a single algorithm, this kind of approach requires custom tuning that will differ greatly across different algorithms. This can require a great deal of time to construct. A more general solution can avoid this problem of custom tuning by using examples of what an algorithm could look like based on actual compiled code, and then comparing how similar these examples are to a given function to produce a series of “best guesses” at what the function could be. This approach will not be able to produce a yes or no answer, but the tradeoff for greater flexibility is the advantage we were aiming for.

Generic Algorithm Analysis Solution: Signature Vector Approach

One of the goals we wanted to achieve for this project was to develop a solution that could work for any arbitrary algorithm in any arbitrary binary. While developing a custom fit approach for a specific algorithm will more accurately and definitively be able to detect a single algorithm, this kind of approach requires custom tuning that will differ greatly across different algorithms. This can require a great deal of time to construct. A more general solution can avoid this problem of custom tuning by using examples of what an algorithm could look like based on actual compiled code, and then comparing how similar these examples are to a given function to produce a series of “best guesses” at what the function could be. This approach will not be able to produce a yes or no answer, but the tradeoff for greater flexibility is the advantage we were aiming for.

The signature vector is our first attempt at a general solution. This solution is comprised of two main components: creating an example signature to store for future comparison and using the collection of stored examples to evaluate the functions of a binary executable. The creation of an example produces a file representing the signature in a collection directory. The comparison of examples outputs a list of functions in the evaluated binary text segment with signatures that match each function above a predefined threshold.

Our signature vector approach is named as such because a vector is the data structure used to represent the signature of an evaluated function. A Python dictionary serves as a convenient structure for this purpose. A string is used as a key to index into the dictionary and a component value is returned. This removes the need for a predetermined order as long as the keys are universal across signatures. The components of this vector are all positive values that represent attributes of the function.

The majority of the vector is made up of counts of individual assembly instructions. For example, if a given function has five “add” instructions, the vector would have an “add” component with a value of five. Only instructions that are present in the function will be added as components of its corresponding signature vector, resulting in a vector of varying size across different signatures. The instruction itself is used as the key of its count value in the vector, serving as a convenient way to guarantee consistency.

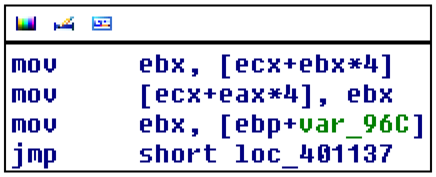
In addition to a series of instruction counts, two more attributes were added to provide a more information about the function being evaluated. Using a graph structure to represent the flow of a function, we generated a count of back edges in the graph and a ratio of code blocks to edges. Back edges can indicate loops while a block-to-edge ratio can indicate branching frequency. We felt that both of these values say something significant about the structure of a function that wouldn’t be obtainable by only counting instructions. These two attributes were assigned unique labels to serve as their keys to the signature vector.

There are some values that we’ve decided against tracking as we developed this solution. The count of “mov” instructions is one such value. This instruction is incredibly common among all assembly functions and the count itself in a single function can overwhelm the other dimensions of its vector, resulting in possibly skewed data. We also considered tracking a simple block count and edge count, but quickly decided against it as it was determined that a count of blocks or edges does not say anything significant. For example, two functions consisting of ten blocks each could be doing wildly different things despite having an equal block count. This consideration was later morphed into the block-to-edge ratio currently being used.

Once the format of the signature vector had been defined, we then looked towards how to compare two different signature vectors. We came across a very convenient method that we could use called cosine comparison. This method essentially measures the angle between two vectors of arbitrary dimension and produces a value indicating how far apart they are. Due to our signature vectors consisting of components greater than or equal to zero, the result is conveniently constrained to a value between zero and one. A one indicates a perfect match while a zero indicates a complete difference. Any value between zero and one indicates a measure of similarity. This similarity value is ultimately the result of the solution to determine function matches.

Generic Algorithm Analysis Solution: Small Primes Product

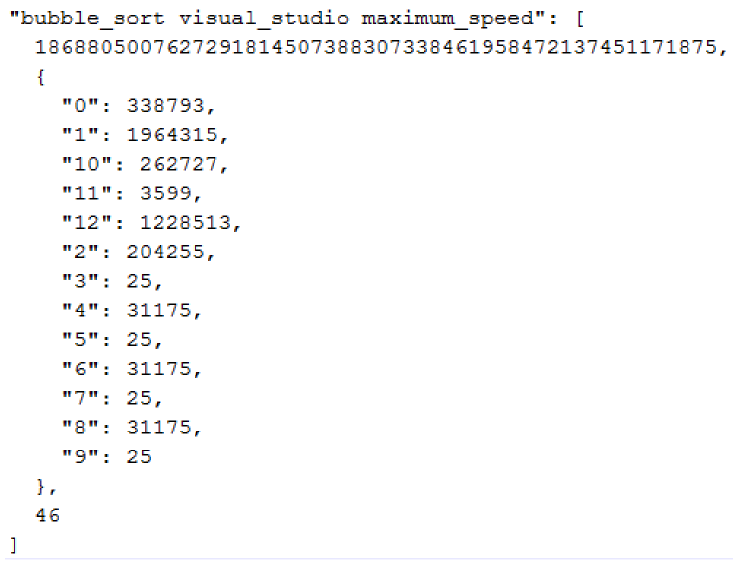
We can uniquely identify functions (and similarly basic blocks) using the Small Primes Product or SPP discussed in Dullen, 2006. The basic approach to this algorithm is to uniquely assign a small prime number starting at 3 for every unique operation, then calculate the product of all the primes. In essence if we look at Figure 2a we have two unique operations, a mov and a jmp.



```
mov     ebx, [ecx+ebx*4]
mov     [ecx+eax*4], ebx
mov     ebx, [ebp+var_96C]
jmp     short loc_401137
```

Figure 2a: Basic block

If mov was assigned the prime number of 3 and jmp was assigned the prime of 5 our SPP for this block would be $3 * 3 * 3 * 5 = 135$. We take the product of all the SPP values for each block within a function to calculate that functions SPP. We store this information inside of a Python dictionary which is exported as a JSON object (Figure 2b) where the key is the algorithm name and the value is the SPP, a list with block ids and their respective SPP, and the count of operations in that function.



```
"bubble_sort visual_studio maximum_speed": [
  1868805007627291814507388307338461958472137451171875,
  {
    "0": 338793,
    "1": 1964315,
    "10": 262727,
    "11": 3599,
    "12": 1228513,
    "2": 204255,
    "3": 25,
    "4": 31175,
    "5": 25,
    "6": 31175,
    "7": 25,
    "8": 31175,
    "9": 25
  },
  46
]
```

Figure 2b: Example JSON object

SPP values are meant to find exact matches between functions regardless of code ordering. This makes SPP very robust to code reordering and some levels of polymorphism. We have added other ways to match similar algorithms. Since SPP is built to match two exact functions we thought we could add some modifiers to match with a predefined threshold. With this approach we are able to develop three different ways to match using SPP values. First method is exact matching in which two functions have to have the same SPP values. Second method is threshold matching in which we generate a threshold amount of additional or missing operations that can occur within a function. This is done by getting the total number of operations for a profile within the JSON object called numOps. We also find the average value of all primes within the dictionary called avg Prime. We then apply the formula $ceil(numOps * threshold) * avgPrime$ where threshold is a number between 1 and 0. The larger the threshold the more operations that can differ. This will give us a lower bound and upper bound for the amount of differences we can have. Our final approach is to match blocks. We have a threshold set on what percentile of blocks must match at minimum. We cycle between all blocks within our profile and all blocks in the function we want to match and compare the SPP values. A match will only occur if the SPP values are the same. Once we have a match we make sure that we do not match that block again. We only state there is a match between two functions if our minimum threshold is met.

Testing Method

Testing our plugins was done by running our scripts on a group of binaries we had compiled, in addition to the binaries supplied on SAGE. Below is a table containing the binaries we tested and their description.

Compiler	**Optimization**	**Algorithm**	**Comment**
Visual Studio	Max Speed	Bubble Sort	Algorithm embedded in main
Visual Studio	Min Size	Bubble Sort	Algorithm embedded in main
Visual Studio	Unoptimized	Bubble Sort	Algorithm in a subroutine

G++	Min Size	Bubble Sort	Algorithm in a subroutine
G++	Unoptimized	Bubble Sort	Algorithm in a subroutine
Visual Studio	Min Size	Bubble Sort	Algorithm embedded in large binary
Unknown	Unknown	Bubble Sort	Binary from SAGE
Unknown	Unknown	Crypto Lib	Binary from SAGE
Visual Studio	Max Speed	Insertion Sort	Algorithm embedded in main
Visual Studio	Min Size	Insertion Sort	Algorithm embedded in main
Visual Studio	Unoptimized	Insertion Sort	Algorithm in a subroutine
G++	Min Size	Insertion Sort	Algorithm in a subroutine
G++	Unoptimized	Insertion Sort	Algorithm in a subroutine
Visual Studio	Max Speed	LCM	Two algorithms GCD and LCM
Visual Studio	Min Size	LCM	Two algorithms GCD and LCM
Visual Studio	Unoptimized	LCM	Two algorithms GCD and LCM
G++	Min Size	LCM	Two algorithms GCD and LCM
G++	Unoptimized	LCM	Two algorithms GCD and LCM
Visual Studio	Max Speed	QuickSort	Algorithm in a subroutine
Visual Studio	Min Size	QuickSort	Algorithm in a subroutine
Visual Studio	Unoptimized	QuickSort	Algorithm in a subroutine
G++	Min Size	QuickSort	Algorithm in a subroutine
G++	Unoptimized	QuickSort	Algorithm in a subroutine

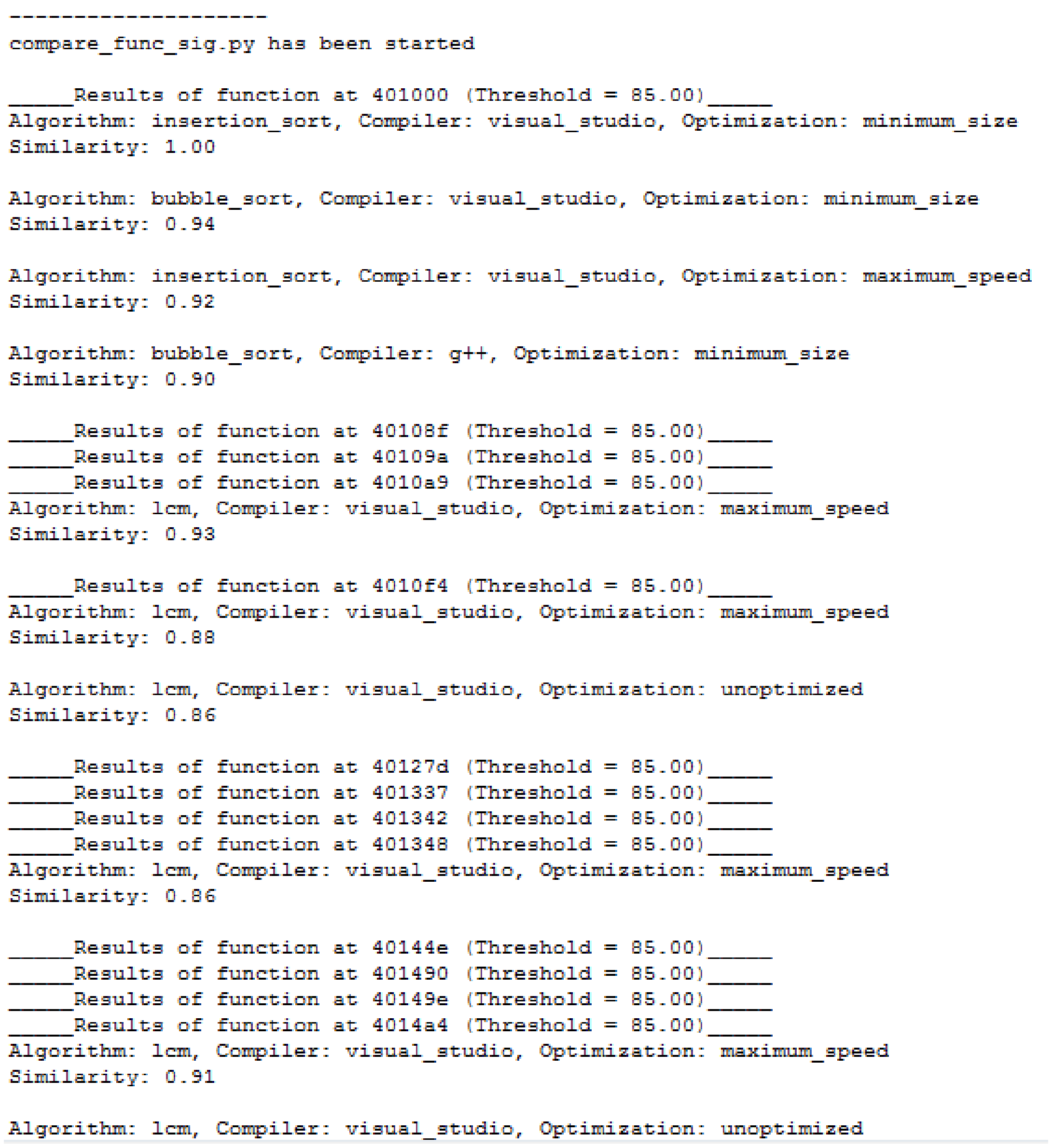
Table 1: Binaries used for testing

Specific Algorithm Testing: Bubble Sort

This plugin was tested by running the bubble_sort_detector.py script against all the binaries listed in Table 1. This script only returned if there was a 100% match. Overall this method was found to be 100% effective in detecting Bubble Sort in all the binaries where the Bubble Sort algorithm was present. We also found no false positive when testing the all the binaries above.

Specific Algorithm Testing: Signature Vector

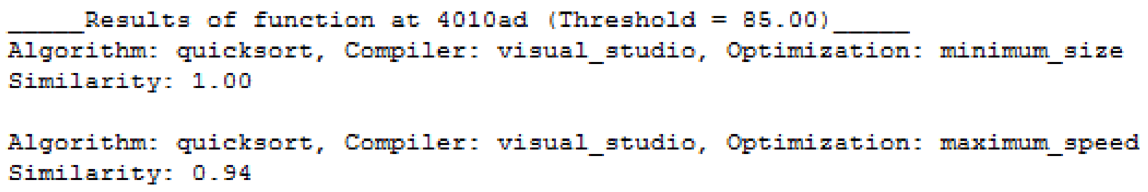
This approach did not worked out as we had hoped. Testing was done by generating a small collection of example signatures from binaries we created to perform different algorithms under different compilers and optimizations. The comparison was then run on some of these same binaries to ensure that it was finding the correct algorithm. A problem was apparent when false positives were being generated frequently from some of the other auto-generated functions in the binaries, resulting in some inaccurate results.



```
--------------------
compare_func_sig.py has been started

_____Results of function at 401000 (Threshold = 85.00)_____
Algorithm: insertion_sort, Compiler: visual_studio, Optimization: minimum_size
Similarity: 1.00

Algorithm: bubble_sort, Compiler: visual_studio, Optimization: minimum_size
Similarity: 0.94

Algorithm: insertion_sort, Compiler: visual_studio, Optimization: maximum_speed
Similarity: 0.92

Algorithm: bubble_sort, Compiler: g++, Optimization: minimum_size
Similarity: 0.90

_____Results of function at 40108f (Threshold = 85.00)_____
_____Results of function at 40109a (Threshold = 85.00)_____
_____Results of function at 4010a9 (Threshold = 85.00)_____
Algorithm: lcm, Compiler: visual_studio, Optimization: maximum_speed
Similarity: 0.93

_____Results of function at 4010f4 (Threshold = 85.00)_____
Algorithm: lcm, Compiler: visual_studio, Optimization: maximum_speed
Similarity: 0.88

Algorithm: lcm, Compiler: visual_studio, Optimization: unoptimized
Similarity: 0.86

_____Results of function at 40127d (Threshold = 85.00)_____
_____Results of function at 401337 (Threshold = 85.00)_____
_____Results of function at 401342 (Threshold = 85.00)_____
_____Results of function at 401348 (Threshold = 85.00)_____
Algorithm: lcm, Compiler: visual_studio, Optimization: maximum_speed
Similarity: 0.86

_____Results of function at 40144e (Threshold = 85.00)_____
_____Results of function at 401490 (Threshold = 85.00)_____
_____Results of function at 40149e (Threshold = 85.00)_____
_____Results of function at 4014a4 (Threshold = 85.00)_____
Algorithm: lcm, Compiler: visual_studio, Optimization: maximum_speed
Similarity: 0.91

Algorithm: lcm, Compiler: visual_studio, Optimization: unoptimized
```

Figure 3a: Comparison ran on insertion sort binary, partial output

The only function in Figure 3a that should have had any real results is the first one. The other functions should not be generating high-similarity matches at all. But, it does seem to be somewhat accurate when looking at a function assembly code. Bubble sort and insertion sort are almost the same algorithm, therefore the four matches in the top function result makes sense. This trend seems to be constant throughout other test runs.



```
_____Results of function at 4010ad (Threshold = 85.00)_____
Algorithm: quicksort, Compiler: visual_studio, Optimization: minimum_size
Similarity: 1.00

Algorithm: quicksort, Compiler: visual_studio, Optimization: maximum_speed
Similarity: 0.94
```

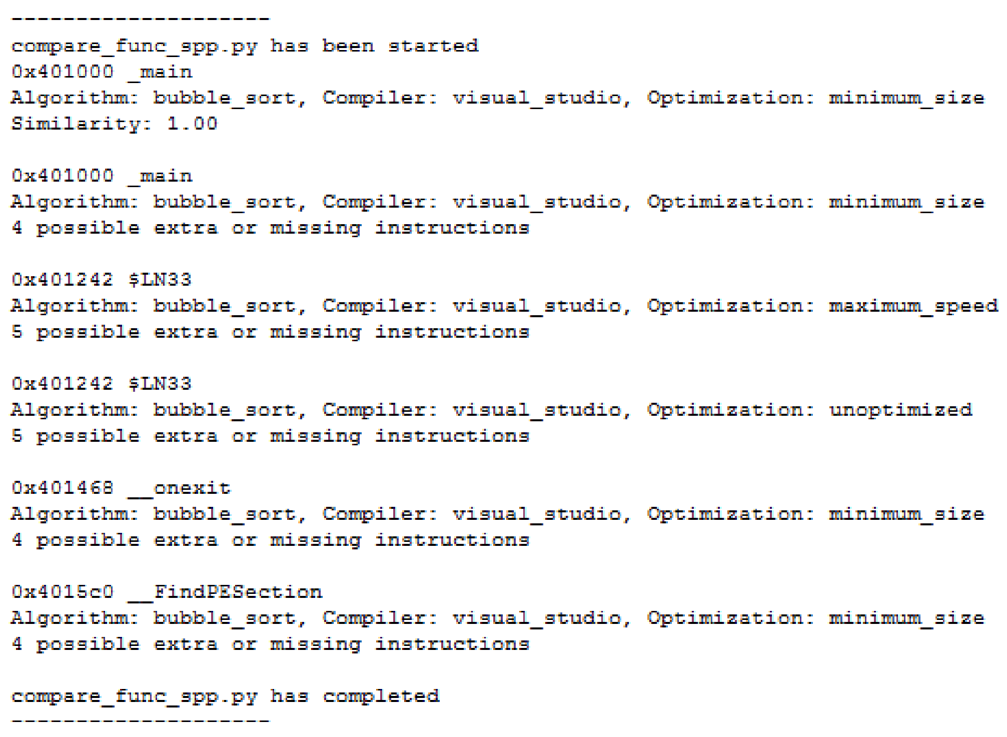
Figure 3b: Comparison ran on quicksort, partial output

In Figure 3b, the function that actually was quicksort obtained accurate results without any unexpected false positives. Results like this seem to show that the idea behind this approach is on the right track, but that the information being collected is not sufficient to uniquely and accurately label an algorithm. It could simply be the case that instruction counts are not enough to create vectors with a significant difference in angle.

Beyond these problems, there are others that we've been aware of before testing. One of the largest roadblocks of this approach is the assumption that a single function represents a single algorithm. For a simple or artificial scenario, this assumption does not cause a significant problem, but it is very likely that this assumption won't hold true for a larger and more complex binary. The result is that this approach cannot detect algorithms embedded within other chunks of code unless an example of that specific instance of code structure was provided beforehand. This approach could still work if the code being evaluated was something that was known to change slightly over time, a quality common to different kinds of malware, with example signatures being maintained of past versions, but it would be impractical for more arbitrary kinds of algorithm detection.

Specific Algorithm Testing: Small Primes Product

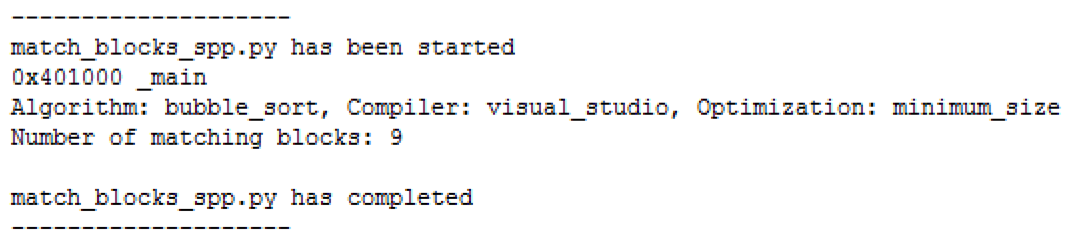
Since our Small Primes Product or SPP plugin matches the operations used in each function (and basic block) the idea of false positives is none existent. With our own permutation of SPP we force SPP to determine if a function (or block) is “functionally” close to another. This produces a slight amount of false positives but changing thresholds helps reduce the amount of false positives. Looking at Figure 4a we can see that we that we have a match. We have also found some other functions that could have a possible match.



```
--------------------
compare_func_spp.py has been started
0x401000 _main
Algorithm: bubble_sort, Compiler: visual_studio, Optimization: minimum_size
Similarity: 1.00

0x401000 _main
Algorithm: bubble_sort, Compiler: visual_studio, Optimization: minimum_size
4 possible extra or missing instructions

0x401242 $LN33
Algorithm: bubble_sort, Compiler: visual_studio, Optimization: maximum_speed
5 possible extra or missing instructions

0x401242 $LN33
Algorithm: bubble_sort, Compiler: visual_studio, Optimization: unoptimized
5 possible extra or missing instructions

0x401468 __onexit
Algorithm: bubble_sort, Compiler: visual_studio, Optimization: minimum_size
4 possible extra or missing instructions

0x4015c0 __FindPESection
Algorithm: bubble_sort, Compiler: visual_studio, Optimization: minimum_size
4 possible extra or missing instructions

compare_func_spp.py has completed
--------------------
```

Figure 4a: Bubble sort minimum size visual studio binary vs threshold matching

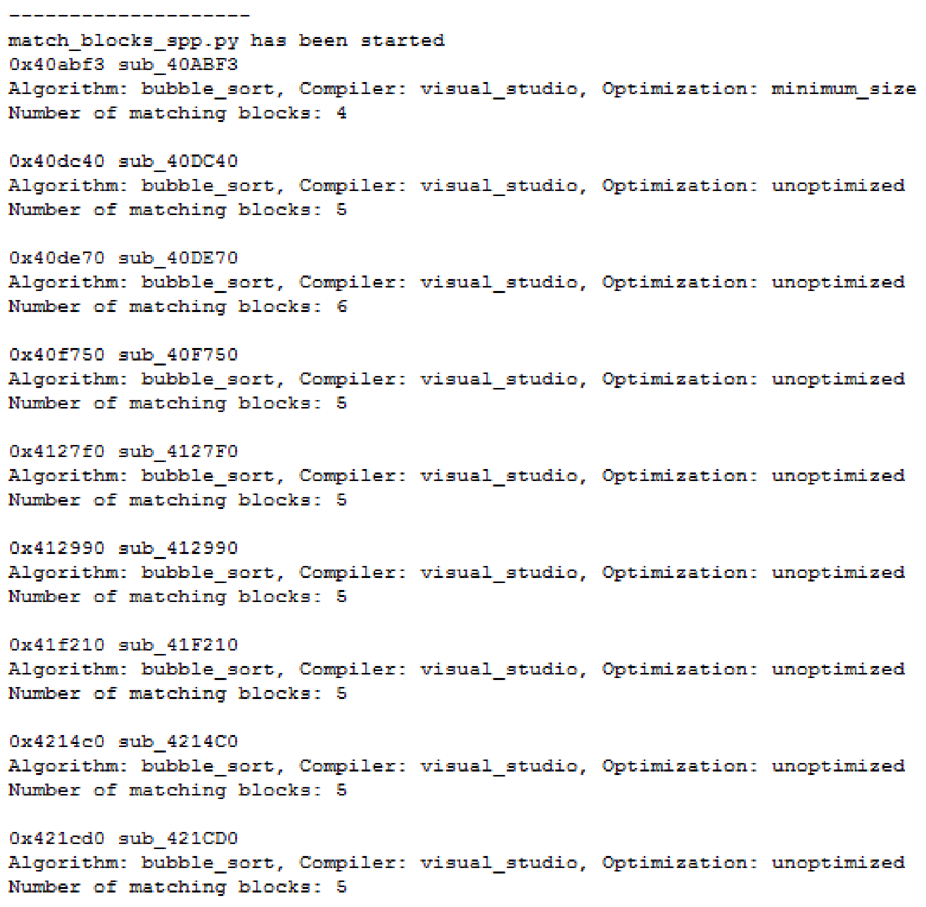
If we run the same binary and try to match by blocks (Figure 4b) we can see that main matches 9 blocks (all blocks) from minimum size profile with no false positives.



```
--------------------
match_blocks_spp.py has been started
0x401000 _main
Algorithm: bubble_sort, Compiler: visual_studio, Optimization: minimum_size
Number of matching blocks: 9

match_blocks_spp.py has completed
--------------------
```

Figure 4b: Bubble sort minimum size visual studio binary vs matching blocks

If we run our matching blocks plugin against a large binary that has bubble sort embedded inside of it compiled in visual studio optimized for minimum size we get a match but there are also some other false positives (Figure 4c).



```
--------------------
match_blocks_spp.py has been started
0x40abf3 sub_40ABF3
Algorithm: bubble_sort, Compiler: visual_studio, Optimization: minimum_size
Number of matching blocks: 4

0x40dc40 sub_40DC40
Algorithm: bubble_sort, Compiler: visual_studio, Optimization: unoptimized
Number of matching blocks: 5

0x40de70 sub_40DE70
Algorithm: bubble_sort, Compiler: visual_studio, Optimization: unoptimized
Number of matching blocks: 6

0x40f750 sub_40F750
Algorithm: bubble_sort, Compiler: visual_studio, Optimization: unoptimized
Number of matching blocks: 5

0x4127f0 sub_4127F0
Algorithm: bubble_sort, Compiler: visual_studio, Optimization: unoptimized
Number of matching blocks: 5

0x412990 sub_412990
Algorithm: bubble_sort, Compiler: visual_studio, Optimization: unoptimized
Number of matching blocks: 5

0x41f210 sub_41F210
Algorithm: bubble_sort, Compiler: visual_studio, Optimization: unoptimized
Number of matching blocks: 5

0x4214c0 sub_4214C0
Algorithm: bubble_sort, Compiler: visual_studio, Optimization: unoptimized
Number of matching blocks: 5

0x421cd0 sub_421CD0
Algorithm: bubble_sort, Compiler: visual_studio, Optimization: unoptimized
Number of matching blocks: 5
```

Figure 4c: Partial output of matching blocks plugin against embedded algorithm

Future Work

Overall our most successful idea was the direct solution approach, we believe expanding on this idea could eventually lead to generic solution that could find multiple different algorithms. This can be accomplished by creating multiple different modules that define very basic program functionality. For example, in Bubble Sort the key functionality we defined was a compare and the corresponding swap. By defining basic program functionality of different algorithms we could create multiple signatures to look for in the binary. For example, our Bubble Sort signature would be {Loop:Loop:Compare:Swap:EndLoop:EndLoop}, or if we define an Insertion Sort signature it could be {Loop:Save First Value:Loop:Move Adjacent Values: EndLoop:Place Saved Value: EndLoop}. Overall the most challenging aspect of this approach is defining, and detecting key functionality that exists in different algorithms you would like to detect. Once this is complete a signature can be trivially made, and used to detect the algorithms in binaries regardless of compiler optimizations.

We had great success with our Small Primes Product or SPP implementations and its permutations. Further expanding on these we can start “learning” more algorithms and seeing how it fairs upto known binaries. One aspect of SPP that can be expanded from this is taking the GCD of two SPP values that are closely related and determining what operations are missing by factoring the GCD from each SPP and then factoring the results into its prime factors. This will allow us to calculate how the two instruction’s sets differ.

Final Thoughts

From our results and information gathered during the duration of this project, we have determined that even though the problem presented to us has no finite solution it is possible to find solutions that are resistant to compiler optimizations and changes. In conclusion, we expect that expanding on plugins explained in this paper could yield a highly accurate generic solution.

The implications of being able to find algorithms in binaries will be very dramatic in the reverse engineering and malware analysis fields. A working solution will speed up the analysis of binaries, allowing users to focus on the bigger picture instead of reversing algorithms previously analysed.

References

Carrera, Ero. (2010). Automated Structural Classification of Malware. Retrieved Mar 3rd, 2013, from http://www.sourceconference.com/publications/bos08pubs/carrera-AutomatedStructuralMalwareClassification.pdf

Dullien, Thomas. (2010). Automated Attacker Correlation for Malicious Code. Retrieved February 20th 2013, from http://ftp.rta.nato.int/public//PubFullText/RTO/MP/RTO-MP-IST-091///MP-IST-091-26.pdf

Dullien, Thomas. (2006). Graph-based comparison of Executable Objects. Retrieved March 3rd, 2013. from http://actes.sstic.org/SSTIC05/Analyse_differentielle_de_binaires/SSTIC05-article-Flake-Graph_based_comparison_of_Executable_Objects.pdf

Flake, Halvar. (2004). Diff, Navigate, Audit. Three applications of graphs and graphing for security. Retrieved February 18th, 2013, from http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-flake.pdf

Flake, Halvar. (2003). More fun with Graphs. Retrieved March 10th, 2013, from http://www.blackhat.com/presentations/bh-federal-03/bh-fed-03-halvar.pdf

Jung, Carrie. (2013, March). Reverse Engineering. Lecture conducted from Mandiant, West Lafayette, IN.

Portnoy Aaron. (2010). NYU Poly Reverse Engineering Lecture. Retrieved March 10th, 2013, from http://pentest.cryptocity.net/files/reversing/Aaron_Pete_ReverseEngineering1.pdf

Schulman, Andrew. (2005). Finding Binary Clones with Opstrings & Function Digests: Part II. Retrieved February 4th, 2013, from http://www.drdobbs.com/finding-binary-clones-with-opstrings-fu/184406203

Silberman, Peter. Loop Detection. Retrieved March 5th, 2013, from uninformed.org/?v=all&a=2&t=pdf