**Assembly Function Recognition (AFR)**

**in embedded systems**

**independent architecture**

Mr. Matan Avitan

matanavtian100@gmail.com

Prof. Zeev (Vladimir) Volkovich

Nov 2019

This report is submitted in partial fulfilment of the requirement for the degree of MSc in Software Engineer

*Abstract*

Function identification is a fundamental challenge in reverse engineering and binary program analysis. For instance, binary rewriting and control flow integrity rely on accurate function detection and identification in binaries. Although many binary program analyses assume functions can be identified a priori, identifying functions in stripped binaries remains a challenge.

This thesis will address the problem of uncertainty in identification of optimized functions. Functions are simply set of commands that describe to CPU what to do every cycle. This thesis including investigation of the difference between debug and optimize functions. The use of probability model including functions detection with custom features independent architecture.

**Categories and Subject Descriptors** Semantics of Programming Languages: Program analysis; Processors: compilers, code generation;

**Keywords** signatures; function identification; x86; x86-64; static binary analysis; reverse engineering; embedded

Contents

[1. Introduction and research goal statement 5](#_Toc18606155)

[2. Literature Review 8](#_Toc18606156)

[Similarity in binaries 8](#_Toc18606157)

[Dynamic approaches 8](#_Toc18606158)

[Semantic differencing 9](#_Toc18606159)

[Equivalence checking 9](#_Toc18606160)

[Locating compiler bugs 9](#_Toc18606161)

[Structure-based static methods 9](#_Toc18606162)

[Similarity of Binaries through re-Optimization 10](#_Toc18606163)

[Equivalence checking and semantic differencing 10](#_Toc18606164)

[Compiler bug-finding 11](#_Toc18606165)

[Detecting software plagiarism 11](#_Toc18606166)

[n-grams based methods 12](#_Toc18606167)

[Similarity measures developed for code synthesis testing 12](#_Toc18606168)

[Similarity measures for source code 12](#_Toc18606169)

[F.L.I.R.T 12](#_Toc18606170)

[3. Research Approach 14](#_Toc18606171)

[3.1 Feasibility Study and Theory 14](#_Toc18606172)

[Mnemonic match 18](#_Toc18606173)

[Command rare match 18](#_Toc18606174)

[Constant match 19](#_Toc18606175)

[Nested function match 20](#_Toc18606176)

[Jump match 21](#_Toc18606177)

[3.2 Optimization Problem 26](#_Toc18606178)

[3.3 Statistics 31](#_Toc18606179)

[Jump match feature 31](#_Toc18606180)

[Mnemonic match feature 31](#_Toc18606181)

[Mnemonic subsequence match feature 32](#_Toc18606182)

[Number of args match feature 32](#_Toc18606183)

[Nested function match feature 32](#_Toc18606184)

[4. Design 36](#_Toc18606185)

[4.1 Algorithm Implementation 36](#_Toc18606186)

[Iterative algorithm 37](#_Toc18606187)

[Recursive algorithm 38](#_Toc18606188)

[4.2 UML 39](#_Toc18606189)

[USE CASE 39](#_Toc18606190)

[CLASS DIAGRAM 39](#_Toc18606191)

[SEQUENCE DIAGRAM 40](#_Toc18606192)

[4.3 GUI 40](#_Toc18606193)

[5. Conclusion and Results 41](#_Toc18606194)

[6. References 42](#_Toc18606195)

[7. Appendix 45](#_Toc18606196)

[Matlab Code 45](#_Toc18606197)

[Python Code 46](#_Toc18606198)

# Introduction and research goal statement

One of the major challenges in reverse engineering is understanding the purpose of programs written in modern languages at a high level. This may be considered lost because no ability to spend the time analyzing external programs and significant algorithms that the manufacturer has attached to its main programs. Unfortunately, this process should be repeated on the same directory functions time after time in reverse engineering.

Sometimes, the knowledge of a class of library function can greatly facilitate program analysis. This knowledge may be very helpful in similar analyzes. For example, a C ++ function that works with streams usually has nothing to do with the program's main algorithm. Function recognition is an example of a basic backend engineering challenge in binary program analysis. Also, many expressions of this type of challenge can be found in an open source library. Many manufacturers use Open Source library code to develop their products and this is one of the most prominent principles in software engineering called reuse. Open source code is embedded in their products as one or more binary files or as part of the firmware.

When performing analysis on those binaries, it’s possible to find the original assembly code which runs in the manufacturer's product. Usually, the full firmware is divided into several sections of a code called a function, which requires easier and faster code analysis, and therefore recommended to identify library functions very easily without having to read them without spend time on this. Standard library functions, sometimes even up to 95% of all functions are called standard functions.

For instance, one well known compiler, the "Hello, world!" program contains:

* library functions 294
* function main() 1



Figure 1

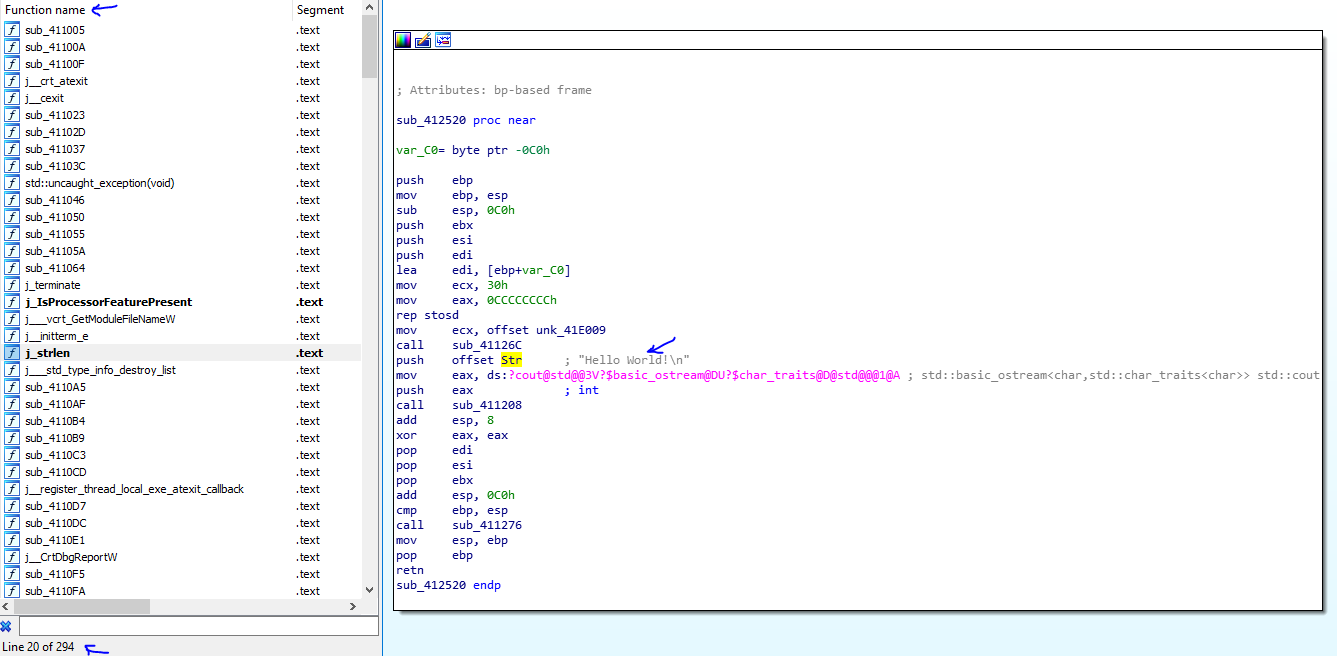


Figure 2

Of course, this is an artificial example, but this analysis showed that the main function contains an average of 50% of open source libraries. Therefore, the researcher was forced to spend more than half of his time researching library functions. An analysis of an unknown plan is like solving a giant crossword puzzle: the more know, the easier it is to guess the next word. During a study, note that meaningful comments and names in the function can provide a faster understanding of the purpose.

There is no intention of trying to achieve a perfect identity, it is impossible in 100% theoretically. Moreover, recognizing certain functions can lead to undesirable consequences. For example, the detection of the next function:

1. push    bp
2. mov     bp, sp
3. xor     ax, ax
4. pop     bp
5. ret

would lead to many misidentifications. It is worth noting that in modern C++ libraries one can find a lot of functions that are identical byte-to-byte but have different names.

As described above, after compiling an entire directory (e.g. libc.so) creating a binary file through which wanted to identify all functions in the binary given by the manufacturer.

For example:

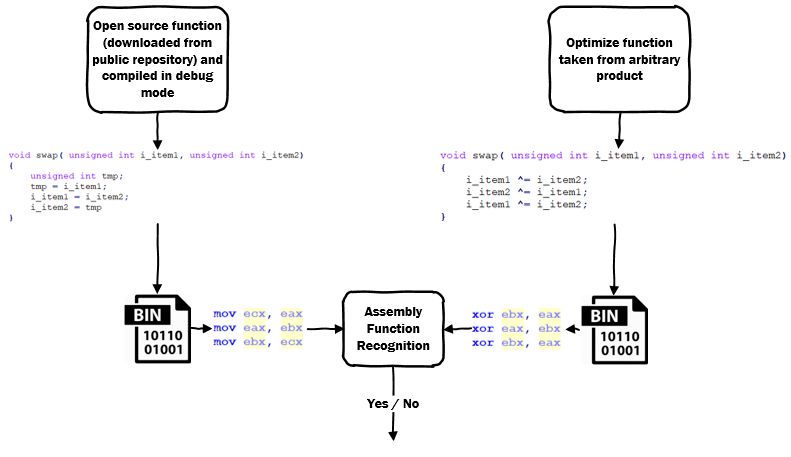


Figure 3

This thesis will answer the following question:

***How to adjust between the assembly optimize function taken from the product and the assembly function is known to compile in a debug environment downloaded from open source repository independent architecture?***

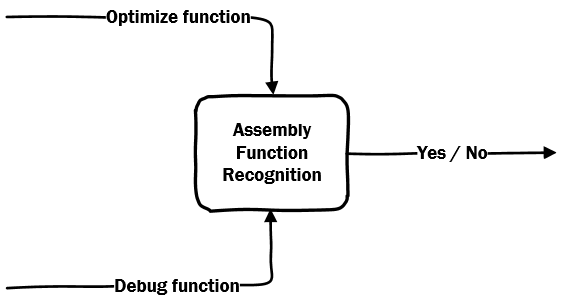


Figure 4

# Literature Review

This section describes relevant works that exist in those areas of interest.

Similarity in binaries

Ng and Prakash [6] operate in a similar context of finding code-reuse and IP theft in binaries, using symbolic execution and theorem provers. They lack a cross-compiler evaluation and use some syntactic heuristics (number of arguments) which affect accuracy. [1] a commonly used commercial tool for comparing whole binary libraries and executables. It performs a many-to-many comparison of all the procedures inside the binary and relies on the connections between these procedures (call-graph) as partial evidence of similarity. BinDiff ignores procedure semantics altogether [1]. The comparison is heavily based on syntactic heuristics, including procedure names (which are unavailable in stripped binaries), the number of jumps, the number of basic blocks, etc. [2] is also heavily biased towards syntax, and therefore fails to handle differences that result from different compilers, as shown by [10]. Pewny et al. [7] uses sampling of SMT-solver simplified basic blocks, lifted using VEX. They report false positives in their results, which can be linked to the sampling-based method because it is not countered by a statistical framework to account for spurious matches from commonly found computations. Furthermore, they require a complex BHB process to be performed online on each pair of compared CFGs (after some pruning). Recently the authors of [15] extracted syntactic and structural features from the code, which hinders.

Dynamic approaches

Egele et al. [3] propose a dynamic method where the procedures are executed using random values and the resulting environment is compared. The reported accuracy is problematic in our scenarios (a tool for a human expert). Aiken et al. [8] find the equivalence of source and machine level code by searching for a simulation relation based on traces of execution. The data driven approach could be applied to our setting of a binary similarity but may not scale due to the computationally heavy search process.

Moreover, dynamic approaches are limited because the executable may not be easily run externally (i.e., it may target a different architecture than the researcher’s machine) and may thus require non-trivial execution access to devices (e.g., IoT devices). Also, running (any part) of the possibly vulnerable library may expose the user to risk.

Semantic differencing

Partush et al. [25, 24] produce an abstract representation of program difference for C programs with loops. The work is limited using costly abstract domains and does not operate on binaries. SymDiff by Lahiri et al. [20] leverages a program verifier to prove equivalence of whole procedures by translating them to BoogieIVL or produce a counter example. It has limited handling for loops and will require a translation of binary to full procedures. Furthermore, the use of provers prevents the approach from scaling.

Equivalence checking

Engler et al. [28] present a symbolic execution approach for proving procedure equivalence for finding bugs in different implementations of the same procedures. They work on the source code level and their method is limited to non-looping code. Lahiri et al. [20] apply a recursion rule to verify the equivalence of recursive functions. The rule operates at the level of source code and relies on a theorem prover, which limits its scalability.

Locating compiler bugs

Hawblitzel et al. [5] compare intermediate language (IL) code produced from different compilers to root-cause compiler bugs, and thus focus on full equivalence instead of similarity. This approach is not likely to scale in our setting, as it uses semantic tools for proving equivalence.

Structure-based static methods

The authors of [28] present an interesting approach for finding similarity using expression trees and their similarity to each other, but this approach is vulnerable to code motion and is not suited to cross-compiler search as the different compilers generate different calculation “shapes” for the same calculation. Jacobson et al. [17] attempt to fingerprint binary procedures using the sequence of system calls used in the procedure. This approach is unstable under patching and is only applicable to procedures which contain no indirect calls or use system calls directly. Eschweiler et al. [14] target multiple architectures

and rely on the (static) control flow structure of the procedure for similarity along with a numeric filtering pre-stage for scalability. This reliance on syntactic features leads to sub-optimal accuracy and limitations, making it not applicable in some cases (such as procedures with a small number of branches).

Horwitz et al. [7] recognized the importance of statistical reasoning, yet their method is geared towards source code and employs n-grams, which were shown to be a weak representation for binary similarity tasks [11]. Jang et al. [18], who also use this basic n-gram decomposition, show a similarity engine, which can be considered complimentary to our hashed search. However, it targets a different scenario of malware detection using features. N-grams are yet again used by Khoo et al. [19], in combination with graphlets. The approach is structural as well, and thus is susceptible to structural changes and further suffers from limitations like [14].

Similarity of Binaries through re-Optimization

Eran Yahav, Nimrod Partush and Yaniv David present a scalable approach for establishing similarity between stripped binaries (with no debug information). The main challenge in binary similarity, is to establish similarity even when the code has been compiled using different compilers, with different optimization levels, or targeting different architectures. Overcoming this challenge, while avoiding false positives, is invaluable to the process of reverse engineering and the process of locating vulnerable code. They present a technique that is scalable and precise, as it alleviates the need for heavyweight semantic comparison by performing out-of-context re-optimization of procedure fragments. It works by decomposing binary procedures to comparable fragments and transforming them to a canonical, normalized form using the compiler optimizer, which enables finding equivalent fragments through simple syntactic comparison. They use a statistical framework built by analyzing samples collected “in the wild” to generate a global context that quantifies the significance of each pair of fragments and uses it to lift pair wise fragment equivalence to whole procedure similarity. They have implemented our technique in a tool called GitZ and performed an extensive evaluation. They show that GitZ can perform millions of comparisons efficiently and find similarity with high accuracy.

Equivalence checking and semantic differencing

[23, 24, 27] are aimed at proving equivalence and describing differences between (versions of) procedures in high-level code, and do not apply to searching for similarity in machine code. The authors of [23, 24] offer some handling for loops, but apply computationally expensive abstract domain libraries, which do not scale in our setting. Sharma et al. [8] present a technique for proving equivalence between high-level code and machine code, with loops, by trying to find a simulation relation between the two. However, the search is computationally demanding and will not scale. Lahiri et al. [19] narrow the scope to assumed-to-be but cannot-be-proved-to-be-equivalent snippets of binary code and attempts to find a set of changes (add or delete) to complete the proof. While having interesting implications for our problem, the assumption of a variable mapping encumbers the transition. Ng and Prakash [6] use symbolic execution for similarity score calculation, but it is not geared towards cross-compiler search, and is limited to handling patched procedures (specifically, only handles one calling convention and rejects procedures based on the number of inputs.)

Compiler bug-finding

Hawblitzel et al. [5] present a technique that handles compiled versions of the same procedure from different compilers. Their goal is to identify root causes of compiler bugs, and their approach cannot be directly applied to our setting as they: (i) require strict equivalence and thus even a slight change would be deemed a bug, (ii) know the IL originated from the same code allowing them to easily match inputs and outputs (i.e. these are labeled) for solving, which is not the case for machine code, and (iii) offer no method for segmenting procedures and thus are limited in handling loops (they use loop unrolling up to 2 iterations).

Detecting software plagiarism

Moss [22] (Measure Of Software Similarity) is an automatic system for determining the similarity of programs. To date, the main application of Moss has been in detecting plagiarism in programming classes. Moss implements some basic ideas that resemble our approach: (i) it decomposes programs and checks for fragment similarity and (ii) it provides the ability to ignore common code. However, in Moss the code is broken down to lines and checked for an exact syntactic match, while Esh decomposes at block level and checks for semantic similarity. Furthermore, in Moss the ignored common code must be supplied by the user as the homework base template code, which is expected to appear in all submissions, while Esh finds common strands by statistical analysis. All the ideas implemented in Moss are preliminary ideas that bear resemblance to ours but are not fully developed in a research paper or evaluated by experiments over binaries.

n-grams based methods

The authors of [17] propose a method for determining even complex lineage for executables. Nonetheless, at its core their method uses linear n-grams combined with normalization steps (in this case also normalizing jumps), is inherently flawed due to reliance on the compiler to make the same layout choices.

Similarity measures developed for code synthesis testing

The authors of [21] propose an interesting way to compare x86 loop-free snippets to perform transformation correctness tests. The similarity is calculated by running the code on selected inputs, and quantifying similarity by comparing outputs and states of the machine at the end of the execution (for example counting equal bits in the output). This distance metric does offer a way to compare two code fragments and possibly to compute similarity, but requires dynamic execution on multiple inputs, which makes it infeasible for our cause.

Similarity measures for source code

There has been a lot of work on detecting similarities in source code (cf. [12]). As our problem deals with binary executables, such approaches are inapplicable. (We discussed an adaptation of these approaches to binaries [9] in Sec. 1). Using program dependence graphs was shown effective in comparing functions using their source code [11] but, applying this approach to assembly code is difficult. Assembly code has no type information, variables are not easily identified [4], and attempting to create a PDG proves costly and imprecise.

F.L.I.R.T

FLIRT stands for Fast Library Identification and Recognition Technology. This technology developed by Hex-Rays and implemented in IDA software. It׳s allows to recognize standard library functions generated by supported compilers and greatly improves the usability and readability of generated disassembly’s. See this Pascal and this Delphi example as well. This tool base on signature of several bytes of functions.

This method based on creating a database of all the functions from all libraries wanted to recognize. Then, check, at each byte of the program being disassembled, whether this byte can mark the start of a standard library function.

The information required by the recognition algorithm is kept in a signature file. Each function is represented by a pattern. Patterns are first 32 bytes of a function where all variant bytes are marked.

For example:

1. 558BEC0EFF7604..........59595DC3558BEC0EFF7604..........59595DC3 \_registerbgidriver
2. 558BEC1E078A66048A460E8B5E108B4E0AD1E9D1E980E1C0024E0C8A6E0A8A76 \_biosdisk
3. 558BEC1EB41AC55604CD211F5DC3.................................... \_setdta
4. 558BEC1EB42FCD210653B41A8B5606CD21B44E8B4E088B5604CD219C5993B41A \_findfirst

# Research Approach

# Feasibility Study and Theory

The first feasibility study starts in two compilation of boost libraries.

First compilation was a debug. To compile as debug, you need to include a debug flag in Linux shell, this command is used as input to the “*make*” program to compile with debug symbols, functions name, variables name, and many more information.

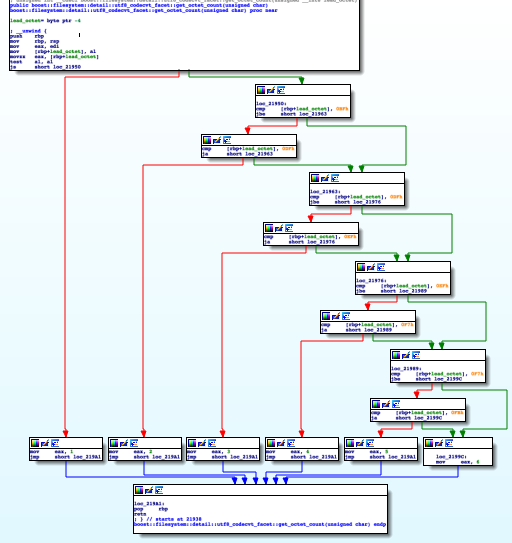


Figure 5

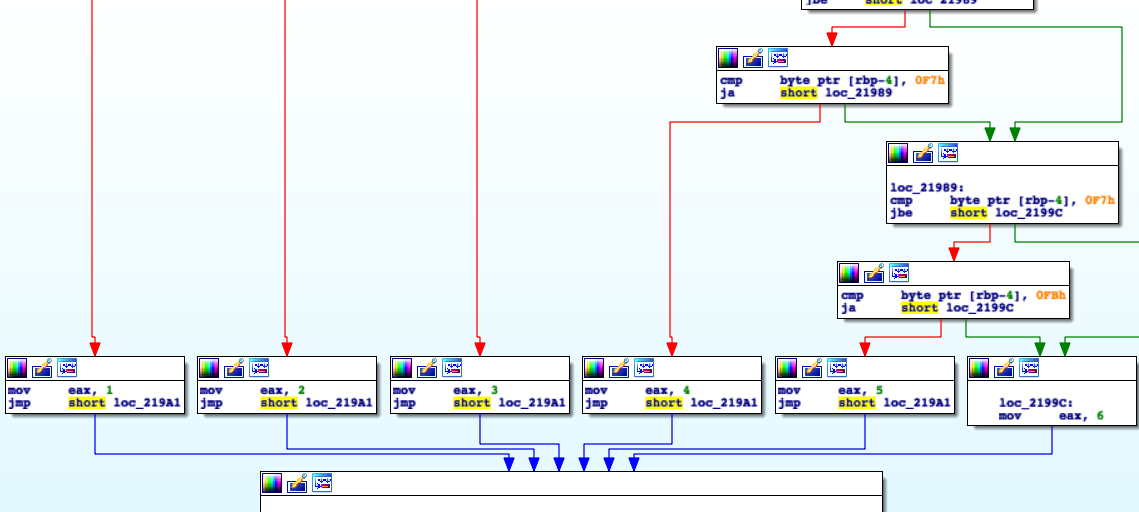


Figure 6

Second compilation was an optimized. As well-known, optimize compilation is tries to minimize or maximize some attributes of an executable computer program. To compile as optimize needed to add the flag “-02” as input to “make” program through boost script. The optimize process drop all information from executable here, each assembly implementation function may be smaller and different than debug implementation.

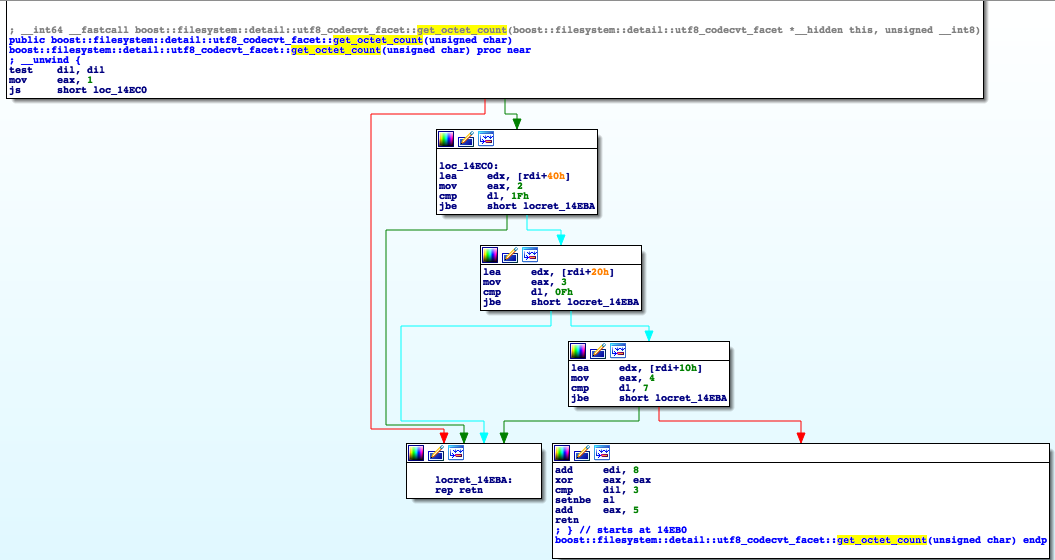


Figure 7

Let׳s take a real example from binaries and run our algorithm to check if its works.

The examples taken from boost libraries. A random way chosen to do comparison between “scoped\_lock(void const volatile\*)” function.

This is the “*scoped\_lock”* function compiled with debug flag:

1. .text:000000000000074A                 push    rbp
2. .text:000000000000074B                 mov     rbp, rsp
3. .text:000000000000074E                 mov     [rbp+**this**], rdi
4. .text:0000000000000752                 mov     [rbp+addr], rsi
5. .text:0000000000000756                 mov     rcx, [rbp+addr]
6. .text:000000000000075A                 mov     rdx, 0C7CE0C7CE0C7CE0Dh
7. .text:0000000000000764                 mov     rax, rcx
8. .text:0000000000000767                 mul     rdx
9. .text:000000000000076A                 shr     rdx, 5
10. .text:000000000000076E                 mov     rax, rdx
11. .text:0000000000000771                 shl     rax, 2
12. .text:0000000000000775                 add     rax, rdx
13. .text:0000000000000778                 shl     rax, 3
14. .text:000000000000077C                 add     rax, rdx
15. .text:000000000000077F                 sub     rcx, rax
16. .text:0000000000000782                 mov     rdx, rcx
17. .text:0000000000000785                 shl     rdx, 6
18. .text:0000000000000789                 lea     rax, boost\_\_atomics\_\_detail\_\_anonymous\_namespace\_\_g\_lock\_pool
19. .text:0000000000000790                 add     rdx, rax
20. .text:0000000000000793                 mov     rax, [rbp+**this**]
21. .text:0000000000000797                 mov     [rax], rdx
22. .text:000000000000079A                 mov     rax, [rbp+**this**]
23. .text:000000000000079E                 mov     rax, [rax]
24. .text:00000000000007A1                 mov     [rbp+var\_10], rax
25. .text:00000000000007A5                 mov     [rbp+var\_20], 2
26. .text:00000000000007AC                 mov     eax, [rbp+var\_20]
27. .text:00000000000007AF                 mov     [rbp+var\_1C], eax
28. .text:00000000000007B2                 cmp     [rbp+var\_1C], 0
29. .text:00000000000007B6                 jz      **short** loc\_7DB
30. .text:00000000000007B8                 cmp     [rbp+var\_1C], 1
31. .text:00000000000007BC                 jz      **short** loc\_7DA
32. .text:00000000000007BE                 cmp     [rbp+var\_1C], 2
33. .text:00000000000007C2                 jz      **short** loc\_7D6
34. .text:00000000000007C4                 cmp     [rbp+var\_1C], 4
35. .text:00000000000007C8                 jz      **short** loc\_7D2
36. .text:00000000000007CA                 cmp     [rbp+var\_1C], 6
37. .text:00000000000007CE                 jnz     **short** loc\_7D9
38. .text:00000000000007D0                 jmp     **short** loc\_7D3
39. .text:00000000000007D2                 nop
40. .text:00000000000007D3                 nop
41. .text:00000000000007D4                 jmp     **short** loc\_7DB
42. .text:00000000000007D6                 nop
43. .text:00000000000007D7                 jmp     **short** loc\_7DA
44. .text:00000000000007D9                 nop
45. .text:00000000000007DA                 nop
46. .text:00000000000007DB                 mov     rdx, [rbp+var\_10]
47. .text:00000000000007DF                 mov     eax, 1
48. .text:00000000000007E4                 xchg    al, [rdx]
49. .text:00000000000007E6                 test    al, al
50. .text:00000000000007E8                 jz      **short** loc\_843
51. .text:00000000000007EA                 pause
52. .text:00000000000007EC                 mov     rax, [rbp+**this**]
53. .text:00000000000007F0                 mov     rax, [rax]
54. .text:00000000000007F3                 mov     [rbp+var\_8], rax
55. .text:00000000000007F7                 mov     [rbp+var\_18], 0
56. .text:00000000000007FE                 mov     eax, [rbp+var\_18]
57. .text:0000000000000801                 mov     [rbp+var\_14], eax
58. .text:0000000000000804                 cmp     [rbp+var\_14], 0
59. .text:0000000000000808                 jz      **short** loc\_82D
60. .text:000000000000080A                 cmp     [rbp+var\_14], 1
61. .text:000000000000080E                 jz      **short** loc\_82C
62. .text:0000000000000810                 cmp     [rbp+var\_14], 2
63. .text:0000000000000814                 jz      **short** loc\_828
64. .text:0000000000000816                 cmp     [rbp+var\_14], 4
65. .text:000000000000081A                 jz      **short** loc\_824
66. .text:000000000000081C                 cmp     [rbp+var\_14], 6
67. .text:0000000000000820                 jnz     **short** loc\_82B
68. .text:0000000000000822                 jmp     **short** loc\_825
69. .text:0000000000000824                 nop
70. .text:0000000000000825                 nop
71. .text:0000000000000826                 jmp     **short** loc\_82D
72. .text:0000000000000828                 nop
73. .text:0000000000000829                 jmp     **short** loc\_82C
74. .text:000000000000082B                 nop
75. .text:000000000000082C                 nop
76. .text:000000000000082D                 mov     rax, [rbp+var\_8]
77. .text:0000000000000831                 movzx   eax, byte ptr [rax]
78. .text:0000000000000834                 test    al, al
79. .text:0000000000000836                 setnz   al
80. .text:0000000000000839                 test    al, al
81. .text:000000000000083B                 jz      loc\_79A
82. .text:0000000000000841                 jmp     **short** loc\_7EA
83. .text:0000000000000843                 nop
84. .text:0000000000000844                 pop     rbp
85. .text:0000000000000845                 retn

And the following is the “*scoped\_lock”* function compiled with optimize (-02) flag

1. .text:0000000000000750                 mov     rax, rsi
2. .text:0000000000000753                 mov     rdx, 0C7CE0C7CE0C7CE0Dh
3. .text:000000000000075D                 mov     ecx, 1
4. .text:0000000000000762                 mul     rdx
5. .text:0000000000000765                 shr     rdx, 5
6. .text:0000000000000769                 lea     rax, [rdx+rdx\*4]
7. .text:000000000000076D                 lea     rax, [rdx+rax\*8]
8. .text:0000000000000771                 sub     rsi, rax
9. .text:0000000000000774                 lea     rax, boost::atomics::detail::`anonymous **namespace**'::g\_lock\_pool
10. .text:000000000000077B                 shl     rsi, 6
11. .text:000000000000077F                 add     rax, rsi
12. .text:0000000000000782                 mov     [rdi], rax
13. .text:0000000000000785                 nop     dword ptr [rax]
14. .text:0000000000000788                 mov     edx, ecx
15. .text:000000000000078A                 xchg    dl, [rax]
16. .text:000000000000078C                 test    dl, dl
17. .text:000000000000078E                 jz      **short** locret\_7A0
18. .text:0000000000000790                 pause
19. .text:0000000000000792                 mov     rax, [rdi]
20. .text:0000000000000795                 movzx   edx, byte ptr [rax]
21. .text:0000000000000798                 test    dl, dl
22. .text:000000000000079A                 jnz     **short** loc\_790
23. .text:000000000000079C                 jmp     **short** loc\_788
24. .text:000000000000079E                 align 20h
25. .text:00000000000007A0                 rep retn

According to the algorithm, needed to find the features to comparison between two functions.

Mnemonic match

Describes the grade given to the criterion that evaluates the mnemonic appearance of the commands in the release to the command state in the debug. Committed enumerate the mnemonic that appear in the release and search for match on their command in the debug. In fact, this feature checks the containment between mnemonic from the assumption that the transition between the debug and release function deletes or unifies commands in order to save run time.

Debug function, include the following commands:

push, **mov**, **mul**, **shr**, **shl**, **add**, **sub**, **lea**, cmp, **jz**, **jnz**, **jmp**, **nop**, **xchg**, **test**, **pause**, **movzx**, setnz, pop, **retn**.

Release function, include the following commands:

mov, mul, shr, lea, sub, shl, add, nop, xchg, test, jz, pause, movzx, test, jnz, jmp, retn

So, the probability is:

Command rare match

Describes the score given to the criterion that evaluates the rare commands in release to the rare commands in the debug. In order must enumerate the mnemonic that appears once and search for match on their command between release and debug. This feature considers special commands such as interrupt calls or privilege command (ring 0).

Note:

1. We will always drop the jump commands because always the target jump will be different, and it will always reduce the probability of imagination.
2. We will always treat registrars as the same variable this grade will related as 0.5 (instead 1).

In Release function, the following mnemonic of commands appears one time:

1. mul     rdx
2. shr     rdx, 5
3. sub     rsi, rax
4. shl     rsi, 6
5. add     rax, rsi
6. nop     dword ptr [rax]
7. xchg    dl, [rax]
8. jz      **short** locret\_7A0 // ignore
9. pause
10. movzx   edx, byte ptr [rax]
11. jnz     **short** loc\_790    // ignore
12. jmp     **short** loc\_788    // ignore
13. retn

At this point needed to search for the similar commands in the debug and calculate the probability:

Constant match

Describes the grade given on the similarity between fixed and mnemonic between the release and the debug. This feature draws a similarity between the functions of an encryption library and/or any other function that uses a special constant to calculate something. The reason of attaching of mnemonic is to reduce noise that can be added unnecessarily. The calculation is performed by collecting all the commands with the constant in the release function and then comparing them to the debug function.

The following command included constants in release function:

1. mov     rdx, 0C7CE0C7CE0C7CE0Dh
2. mov     ecx, 1
3. shr     rdx, 5
4. shl     rsi, 6

**Note**:

We will always treat registrars as the same variable this grade will related as 0.5 (instead 1).

Here is the calculation:

Nested function match

Describes the grade given for similarities in function calls between the debug function and the release function.

It so important to note that this feature assume all nested functions identified therefore, needed to run this in recursive mode.

In this case have no calls therefore the probability will be 1.

To be clearly, the following example describe the situation.

Let׳s look at the example, let's take the “printf” function (from uclibc library), this function included calls to two internal functions.

The following is debug function:

1. push    ebp
2. mov     ebp, esp
3. sub     esp, 18h
4. call    \_\_x86\_get\_pc\_thunk\_ax
5. add     eax, 744ABh
6. lea     edx, [ebp+arg\_4]
7. mov     [ebp+arg], edx
8. mov     edx, [ebp+arg]
9. mov     eax, ds:(stdout\_ptr 0B9C7Ch)[eax]
10. mov     eax, dword ptr ds:(\_GLOBAL\_OFFSET\_TABLE\_ 0B9C7Ch)[eax]
11. sub     esp, 4
12. push    edx             ; arg
13. push    [ebp+format]    ; format
14. push    eax             ; stream
15. call    \_\_GI\_vfprintf
16. add     esp, 10h
17. mov     [ebp+rv], eax
18. mov     eax, [ebp+rv]
19. leave
20. retn

The following is release function:

1. call    sub\_10CE9
2. add     eax, 57F37h
3. sub     esp, 0Ch
4. mov     eax, ds:(stdout\_ptr 88C7Ch)[eax]
5. lea     edx, [esp+0Ch+arg\_4]
6. sub     esp, 4
7. push    edx             ; **int**
8. push    [esp+14h+arg\_0] ; **void** \*
9. push    dword ptr ds:(tbyte\_88C7C 88C7Ch)[eax] ; len
10. call    vfprintf
11. add     esp, 1Ch
12. retn

Debug function, contain two calls:

1. call    \_\_x86\_get\_pc\_thunk\_ax
2. call    \_\_GI\_vfprintf

In release function, contain two calls:

1. call    sub\_10CE9
2. call    vfprintf

previously, detected the call to “sub\_10CE9“ as “\_\_x86\_get\_pc\_thunk\_ax” therefore, P(x) will be:

Jump match

Describes the grade given for similarities in jumps with condition between the debug function and the release function.

This assume based on avoiding jump prediction by using the carry in flag register

The following example finds the minimum of two unsigned numbers: if (b < a) a = b;

1. SUB EBX,EAX
2. SBB ECX,ECX
3. AND ECX,EBX
4. ADD EAX,ECX

The next example chooses between two numbers: if () ;

1. CMP EAX,1
2. SBB EAX,EAX
3. XOR ECX,EBX
4. AND EAX,ECX
5. XOR EAX,EBX

Let’s take an example of “pathconf” function:

The following is debug function:

1. push    ebp
2. mov     ebp, esp
3. push    ebx
4. sub     esp, 64h
5. call    \_\_x86\_get\_pc\_thunk\_bx
6. add     ebx, 4D05Dh
7. mov     eax, [ebp+path]
8. movzx   eax, byte ptr [eax]
9. test    al, al
10. **jnz     short** loc\_6CC46
11. mov     eax, ds:(off\_B9F34 - 0B9C7Ch)[ebx]
12. mov     dword ptr gs:[eax], 2
13. mov     eax, 0FFFFFFFFh
14. jmp     loc\_6CD70
15. cmp     [ebp+name], 13h
16. **ja      short** loc\_6CC5D
17. mov     eax, [ebp+name]
18. shl     eax, 2
19. mov     eax, ds:(off\_9FF94 - 0B9C7Ch)[eax+ebx]
20. add     eax, ebx
21. jmp     eax
22. mov     eax, ds:(off\_B9F34 - 0B9C7Ch)[ebx]
23. mov     dword ptr gs:[eax], 16h
24. mov     eax, 0FFFFFFFFh
25. jmp     loc\_6CD70
26. mov     eax, 7Fh
27. jmp     loc\_6CD70
28. mov     eax, 0FFh
29. jmp     loc\_6CD70
30. mov     eax, 0FFh
31. jmp     loc\_6CD70
32. mov     eax, ds:(off\_B9F34 - 0B9C7Ch)[ebx]
33. mov     eax, gs:[eax]
34. mov     [ebp+save\_errno], eax
35. sub     esp, 8
36. lea     eax, [ebp+st]
37. push    eax
38. push    [ebp+path]
39. call    statfs\_0
40. add     esp, 10h
41. test    eax, eax
42. **jns     short** loc\_6CCE2
43. mov     eax, ds:(off\_B9F34 - 0B9C7Ch)[ebx]
44. mov     eax, gs:[eax]
45. cmp     eax, 26h ; '&'
46. **jnz     short** loc\_6CCD8
47. mov     eax, ds:(off\_B9F34 - 0B9C7Ch)[ebx]
48. mov     edx, [ebp+save\_errno]
49. mov     gs:[eax], edx
50. mov     eax, 0FFh
51. jmp     loc\_6CD70
52. mov     eax, 0FFFFFFFFh
53. jmp     loc\_6CD70
54. mov     eax, dword ptr [ebp+st.st\_rdev+4]
55. jmp     loc\_6CD70
56. mov     eax, 1000h
57. jmp     **short** loc\_6CD70
58. mov     eax, 1000h
59. jmp     **short** loc\_6CD70
60. mov     eax, 0
61. jmp     **short** loc\_6CD70
62. mov     eax, 1
63. jmp     **short** loc\_6CD70
64. mov     eax, 0
65. jmp     **short** loc\_6CD70
66. mov     eax, 0FFFFFFFFh
67. jmp     **short** loc\_6CD70
68. sub     esp, 8
69. lea     eax, [ebp+st]
70. push    eax
71. push    [ebp+path]
72. call    stat\_0
73. add     esp, 10h
74. test    eax, eax
75. **js      short** loc\_6CD48
76. mov     eax, [ebp+st.st\_mode]
77. and     eax, 0F000h
78. cmp     eax, 8000h
79. **jz      short** loc\_6CD4F
80. mov     eax, [ebp+st.st\_mode]
81. and     eax, 0F000h
82. cmp     eax, 6000h
83. **jz      short** loc\_6CD4F
84. mov     eax, 0FFFFFFFFh
85. jmp     **short** loc\_6CD70
86. mov     eax, 1
87. jmp     **short** loc\_6CD70
88. mov     eax, 0FFFFFFFFh
89. jmp     **short** loc\_6CD70
90. mov     eax, 0FFFFFFFFh
91. jmp     **short** loc\_6CD70
92. mov     eax, 20h
93. jmp     **short** loc\_6CD70
94. mov     eax, 0FFFFFFFFh
95. mov     ebx, [ebp+var\_4]
96. leave
97. retn

The following is release function:

1. .text:00046FA3                 push    edi
2. .text:00046FA4                 push    esi
3. .text:00046FA5                 push    ebx
4. .text:00046FA6                 call    sub\_F96A
5. .text:00046FAB                 add     ebx, 41CD1h
6. .text:00046FB1                 sub     esp, 60h
7. .text:00046FB4                 mov     ecx, [esp+6Ch+arg\_0]
8. .text:00046FB8                 mov     edx, [esp+6Ch+arg\_4]
9. .text:00046FBC                 test    ecx, ecx
10. .text:00046FBE                 **js**loc\_4709F
11. .text:00046FC4                 test    edx, edx
12. .text:00046FC6                 mov     eax, 7Fh
13. .text:00046FCB                 **jnz     short** loc\_46FD4
14. .text:00046FCD                 add     esp, 60h
15. .text:00046FD0                 pop     ebx
16. .text:00046FD1                 pop     esi
17. .text:00046FD2                 pop     edi
18. .text:00046FD3                 retn
19. .text:00046FD4                 cmp     edx, 13h
20. .text:00046FD7                 **ja**loc\_47088
21. .text:00046FDD                 mov     eax, ds:(off\_65174 - 88C7Ch)[ebx+edx\*4]
22. .text:00046FE4                 add     eax, ebx
23. .text:00046FE6                 jmp     eax
24. .text:00046FE8                 mov     eax, 0FFFFFFFFh
25. .text:00046FED                 jmp     **short** loc\_46FCD
26. .text:00046FEF                 xor     eax, eax
27. .text:00046FF1                 jmp     **short** loc\_46FCD
28. .text:00046FF3                 mov     eax, 1000h
29. .text:00046FF8                 jmp     **short** loc\_46FCD
30. .text:00046FFA                 mov     eax, 0FFh
31. .text:00046FFF                 jmp     **short** loc\_46FCD
32. .text:00047001                 mov     eax, 20h
33. .text:00047006                 jmp     **short** loc\_46FCD
34. .text:00047008                 sub     esp, 8
35. .text:0004700B                 lea     eax, [esp+74h+var\_64]
36. .text:0004700F                 push    eax
37. .text:00047010                 push    ecx
38. .text:00047011                 call    fstat
39. .text:00047016                 add     esp, 10h
40. .text:00047019                 test    eax, eax
41. .text:0004701B                 **js      short** loc\_46FE8
42. .text:0004701D                 mov     eax, [esp+6Ch+var\_54]
43. .text:00047021                 and     eax, 0F000h
44. .text:00047026                 sub     eax, 6000h
45. .text:0004702B                 test    eax, 0FFFFD000h
46. .text:00047030                 setnz   al
47. .text:00047033                 movzx   eax, al
48. .text:00047036                 neg     eax
49. .text:00047038                 or      eax, 1
50. .text:0004703B                 jmp     **short** loc\_46FCD
51. .text:0004703D                 mov     eax, 1
52. .text:00047042                 jmp     **short** loc\_46FCD
53. .text:00047044                 sub     esp, 8
54. .text:00047047                 mov     esi, ds:(off\_88F30 - 88C7Ch)[ebx]
55. .text:0004704D                 lea     eax, [esp+74h+var\_64]
56. .text:00047051                 mov     edi, gs:[esi]
57. .text:00047054                 push    eax
58. .text:00047055                 push    ecx
59. .text:00047056                 call    fstatfs
60. .text:0004705B                 mov     edx, eax
61. .text:0004705D                 add     esp, 10h
62. .text:00047060                 mov     eax, [esp+6Ch+var\_40]
63. .text:00047064                 test    edx, edx
64. .text:00047066                 **jns**loc\_46FCD
65. .text:0004706C                 cmp     dword ptr gs:[esi], 26h
66. .text:00047070                 mov     eax, 0FFFFFFFFh
67. .text:00047075                 **jnz**loc\_46FCD
68. .text:0004707B                 mov     gs:[esi], edi
69. .text:0004707E                 mov     eax, 0FFh
70. .text:00047083                 jmp     loc\_46FCD
71. .text:00047088                 mov     eax, ds:(off\_88F30 - 88C7Ch)[ebx]
72. .text:0004708E                 mov     dword ptr gs:[eax], 16h
73. .text:00047095                 mov     eax, 0FFFFFFFFh
74. .text:0004709A                 jmp     loc\_46FCD
75. .text:0004709F                 mov     eax, ds:(off\_88F30 - 88C7Ch)[ebx]
76. .text:000470A5                 mov     dword ptr gs:[eax], 9
77. .text:000470AC                 mov     eax, 0FFFFFFFFh

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | ja | jnz | js | jns | jz | Total |
| Debug | 1 | 2 | 1 | 1 | 2 | 7 |
| Release | 1 | 2 | 2 | 1 | 0 | 6 |

Table 1

And then will be:

# Optimization Problem

This phase will translate the problem to the mathematic problem and define the exactly optimize problem.

I created arbitrary formula will reflect the main problem.

My formula based on weights and features. According to section 3.1, it’s very easy to find the features. In addition, everyone can add any custom feature.

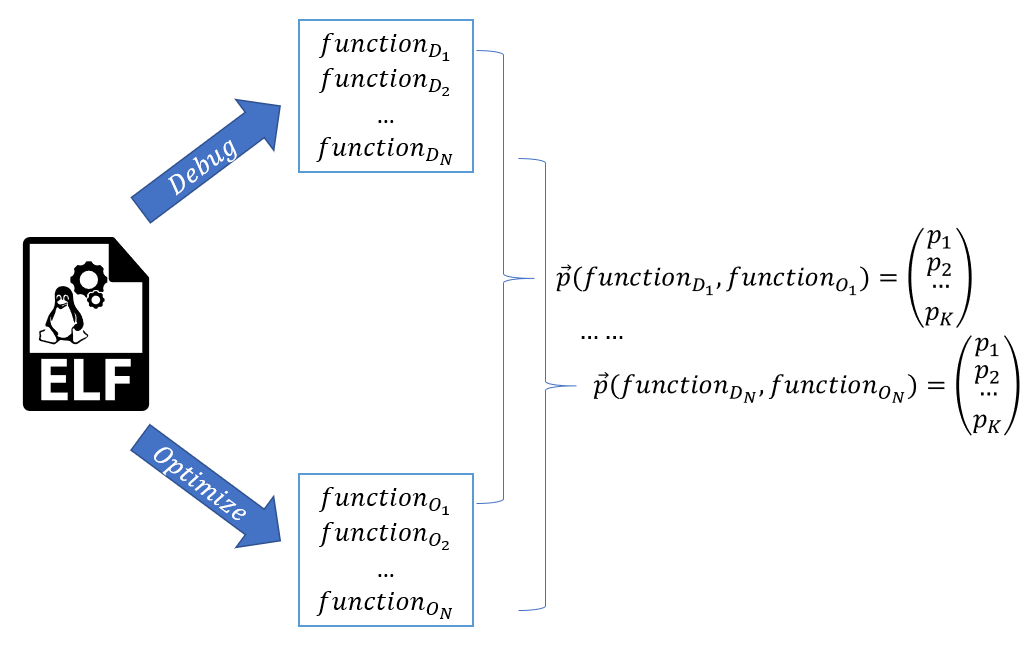


Figure 8

Every and and can be describe in the following matrix:

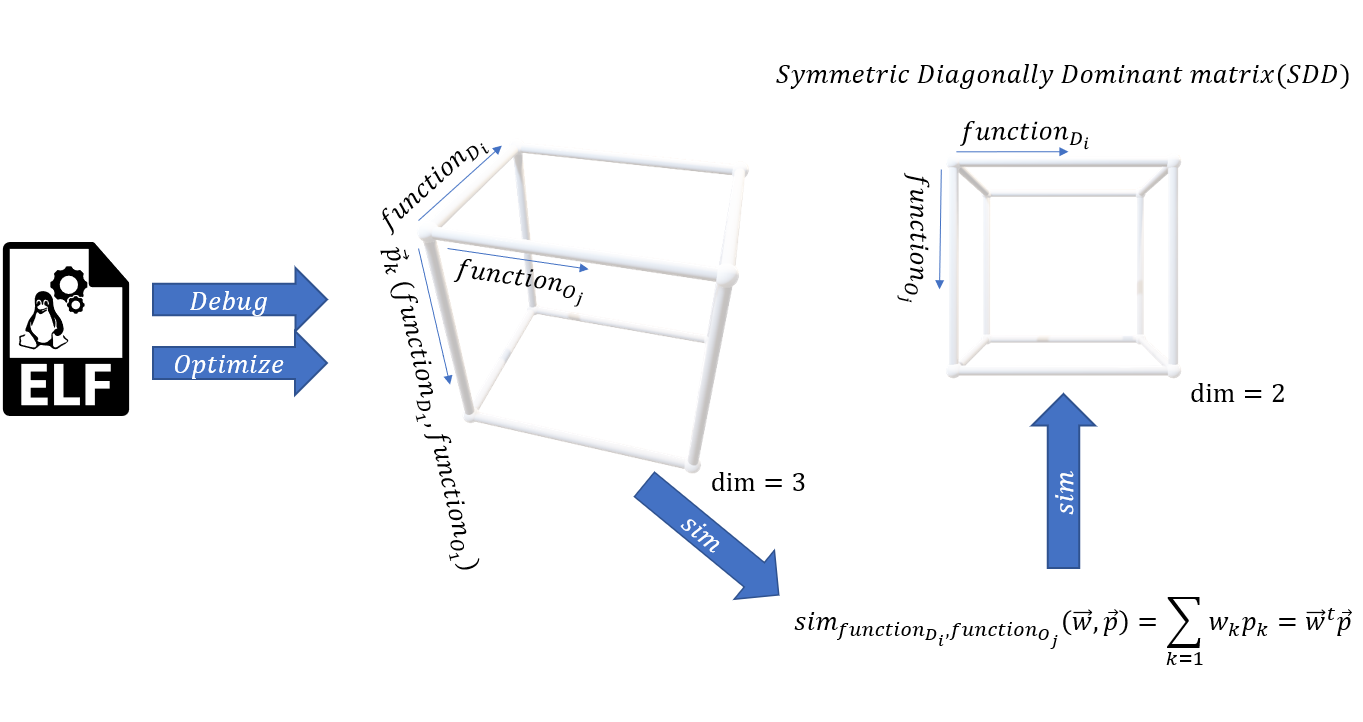


Figure 9

My mission is to find the weights in the following formula:

As can see for every feature have weight partner

As described ([see section 3.1](#_Literature_Review)), can be calculated according to the feature definition.

At this moment, taking the “uclibc” and calculate the features between every pair’s functions (optimize and debug) and will build N formulas (for every pairs function, optimize and debug). So, Lets define our problem:

SDD (Symmetric Diagonally Dominant) define the following:

Note that:

So,

Define:

So,

Define:

Let’s add number of constraints:

* Every weight will be positive
* Every feature will be probabilistic value

These formulas will take account into optimization mathematic problem.

**Note**: maybe this calculation will be relevant per library, this aspect will discuss later

The optimization issue is well defined in each of the following sections. Note that in section (0) and (1), the problem is relatively difficult to resolve and therefore to implement a solution we will use (2).

(0)

(1)

(2)

Such that

It was later discovered that the above theory is correct but inaccurate as it is not very important. The solution needs to be refined to solve the problem precisely. Defining the problem in not handling false positive cases. This means that by defining the problem above, the following condition may occur for several pairs of functions:

So, we will now try to introduce another constraint:

Let define the following parameters:

For every i will be held:

And then we will expand it to:

In general:

Define again:

And,

So,

And finally:

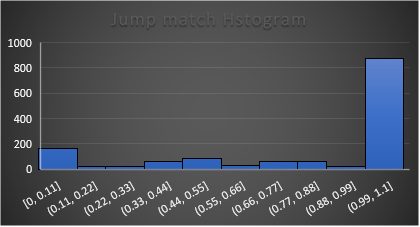
Such that

# Statistics

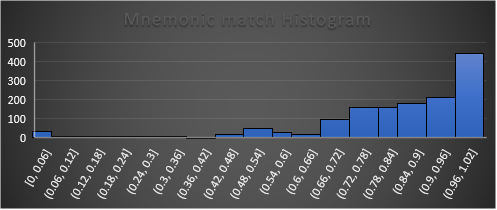
It is highly desirable to quantify the quality of the features presented in previous chapters and therefore, statistics were performed accordingly. In fact, if want to add new features to the model, they should perform initial statistics to ensure there are no errors or noises in the static.

The following is a histogram for each of the features on two identical functions:

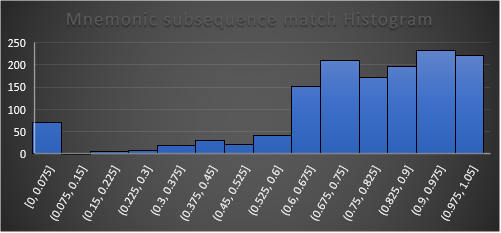
Jump match feature



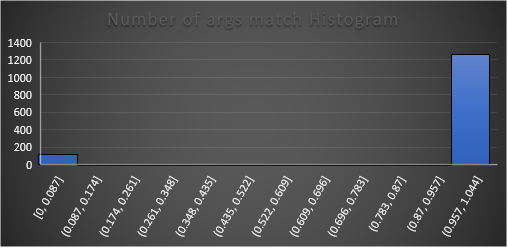
Mnemonic match feature



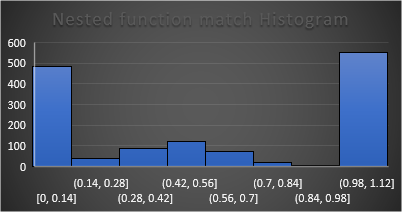
Mnemonic subsequence match feature



Number of args match feature



Nested function match feature



It is very easy to distinguish between good-fit features and information-only features when there is uncertainty.

“Jump match feature” is a feature that can be matched in most cases and so it can be said that it is a quality feature and yet the “Nested function match” feature does not reflect reality and is not sure that it can be taken as part of the calculations.

In any case, we considered every feature that we saw fit, which gave some hint to reality (even partially) and added statistics on adding and downloading features and found that the more features the better the quality of the results.

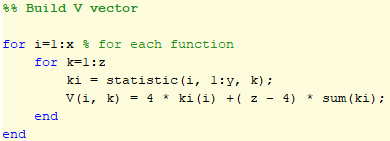
After exporting the statistic (in json format) to MATLAB as a matrix. The data was initially analysed, a code was written to read the statistics and then several arithmetic operations were performed (See the MATLAB code in Appendix section).

The code describing optimization process including:

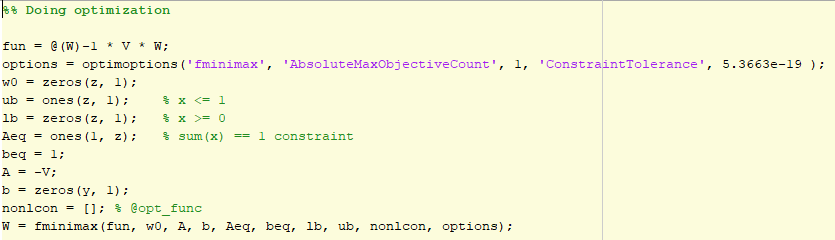
* Building the V vector according to previous section (See “*Optimization problem”* section)

Such that:

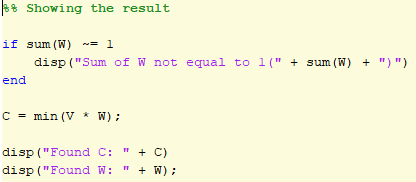
As implemented the following code:



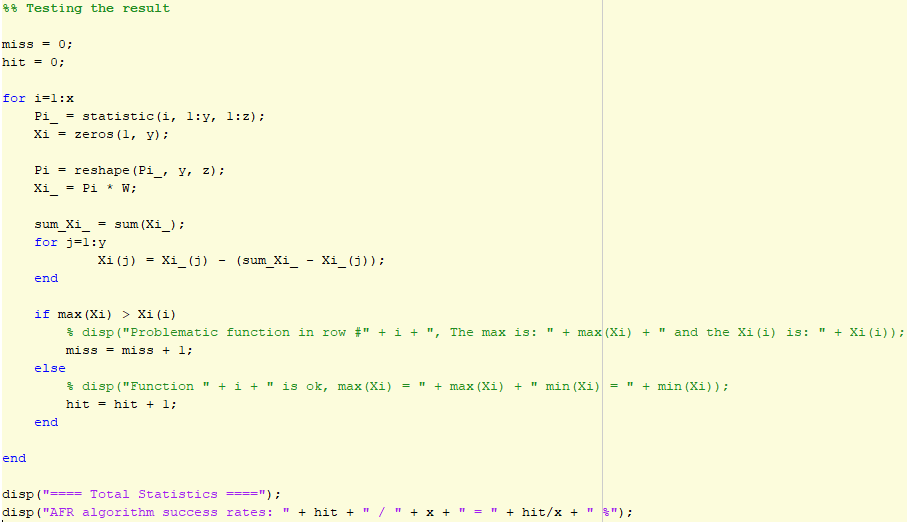
* Mathematical problem definition (See “*Optimization problem”* section). Since MATLAB restricts the use of its functions in noisy functions in the *fminimax* optimization function and we also translated our *maxmin* problem into the *minmax* of MATLAB and the code below:



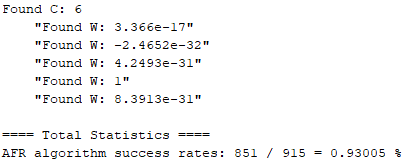
* Showing the result



* Testing the result. Calculating the grade of similarity for each pairs function i and j and check if results equal to real data (needed to get Diagonally Dominant matrix)



Here are the results of running the code:



# Design

The PoC implementation will in python language via IDA software that provide framework to assembly code. IDAPython is an IDA plugin which makes it possible to write scripts for IDA in the Python programming language. IDAPython provides full access to both the IDA API and any installed Python module.

# Algorithm Implementation

The algorithm based on quality of features and weights ([see 3.1 section](#_Literature_Review)).

The propose of the features to identify the similarity between two functions and the propose of the weights to give to every feature percentage of decision.

In this earlier time, can formulate all features that based on code conversion from debug to optimize. Conversion defined to be only in a one way and cannot be reversible. Really, it’s possible to reverse couple of conversions but, some of them very hard to identify and this implemented and described in feature section.

Later, analyzing the weights value need to be to get correct.

That analysis will be based on statistics and histogram graph.

Our algorithm based on few elements that enable to identify function:

* – Describe the **debug** function number as desirable to identify.
* – Describe the **optimize** function number from debug library as desirable to check similarities.
* – Describes the probability of similarities given two functions and . That similarities based on feature number .
* – Contain the primitive functions that not included branches to other internal functions.
* – Contain the non-primitive functions that included branches to other internal functions.
* - Describe the weight given to feature k.
* – indicate the probability to similarity between and .

Iterative algorithm

Given:

)

For each

if is primitive then

add it to

else

add it to

For each in

For each

calculate

append S+=[s]

Max S

mark as similar to

remove and from

For each in

For each

For each feature k

calculate

append S+=[s]

Max S

mark as similar to

remove and from

Recursive algorithm

Given:

)

Build function call graph

Let׳s run DFS algorithm on debug Library

for each in DFS

for each

calculate

append S+=[s]

Max S

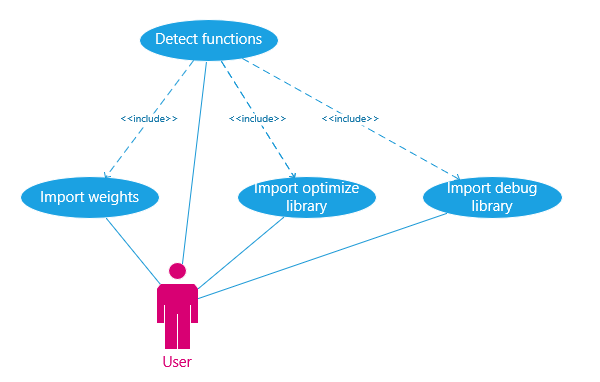
mark as like

mark in three

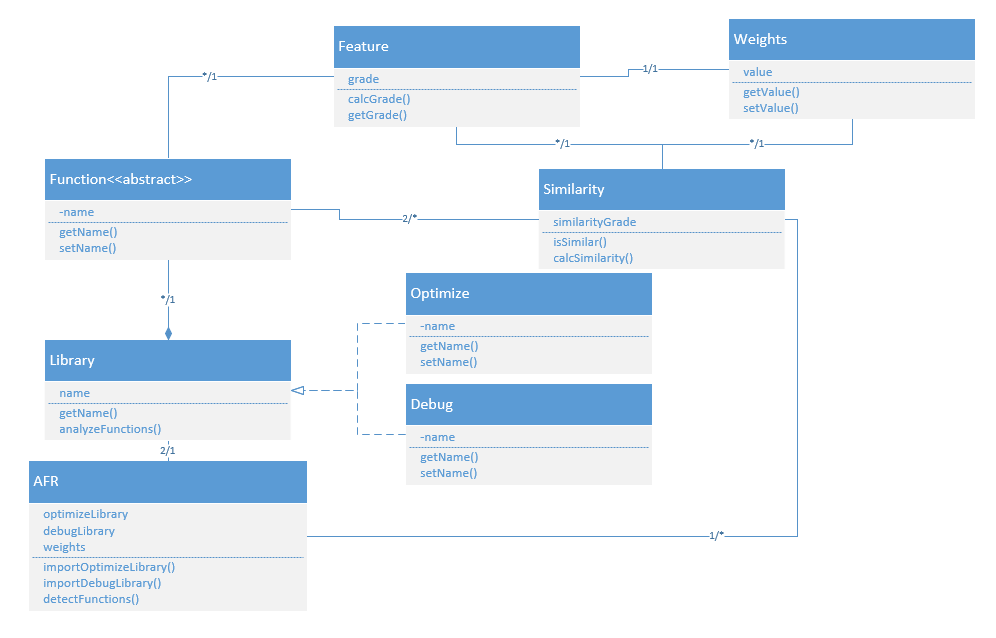
remove from debug

# UML

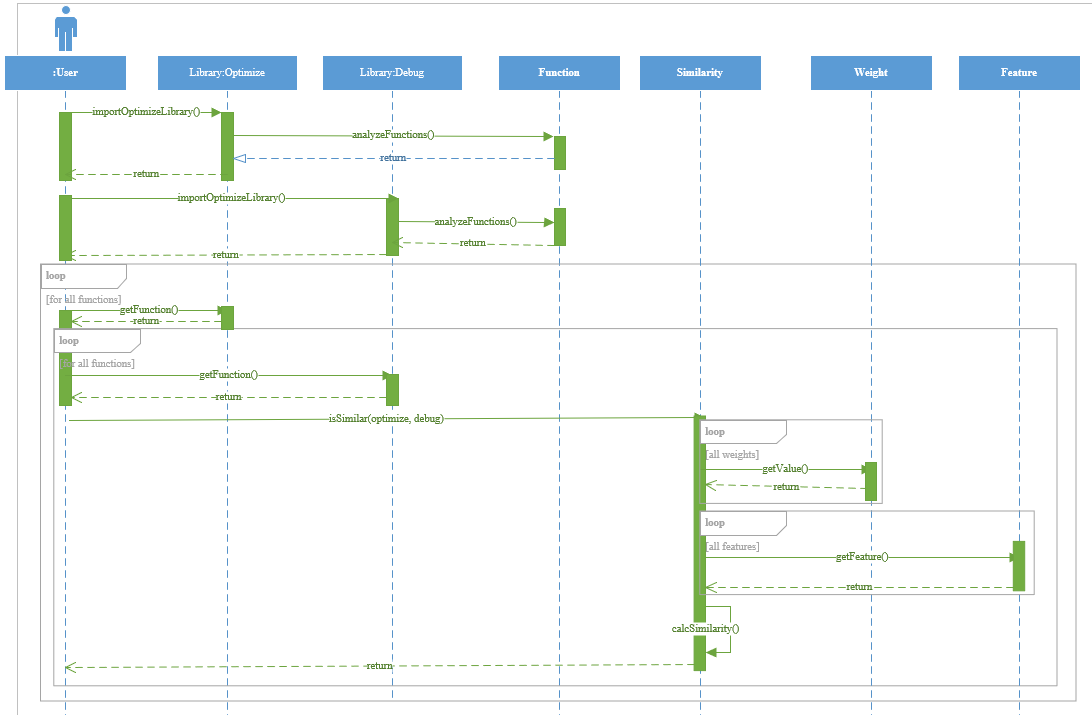
USE CASE



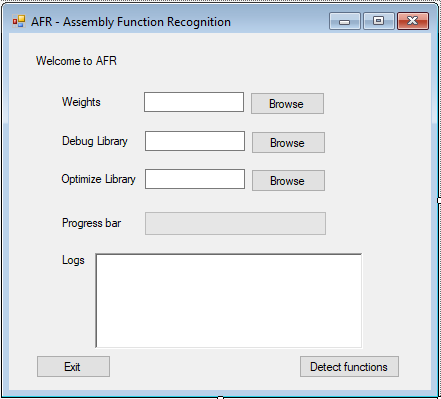
CLASS DIAGRAM



SEQUENCE DIAGRAM



# GUI



# Conclusion and Results

An extensible approach for examining similarities between binary functions was presented. The approach can establish similarity with high precision even when the code is compiled using different compilers, with different optimization levels, or different targeted architectures. The main idea is to score each function based on predetermined features and then find a suitable weight vector that could give importance to each feature. After finding the weight vector, these functions can be identified in binary software packages and save the research process of understanding a function. To determine similarities in the procedure, an engineering statistical model is used that guarantees high identification success. The AFR tool guarantees success at a higher level than the competition and, in addition, AFR is able to make millions of comparisons thoroughly and at a high level of accuracy, and in a faster scale in a competing method.

This thesis leaves several investigations. The following list detail couple of them:

* This model found a solution for functions contain over k assemblies commands (statistics results based on k = 10).

There is another difficulty in identifying functions that contain a small number of commands and so some questions are asked such as what is the minimum k that this model solves? Is it possible to add a feature that solves the lack of information problem in small functions (such as dividing into equivalent groups of k commands) instead of solving the lower block problem?

* It will be interesting to investigate the DNA (C and ) of additional library functions with the same features, this study will yield C and weight values that will be interesting to see the similarities (if exist) and their differences in construction.
* A study of adding new features to assembly code will only improve the model for better results and accuracy.

# References

[1] zynamics bindiff. <http://www.zynamics.com/bindiff.html>.

[2] Y. David and E. Yahav. Tracelet-based code search in executables. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’14, pages 349–360, New York, NY, USA, 2014. ACM.

[3] M. Egele, M.Woo, P. Chapman, and D. Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014., pages 303–317, 2014.

[4] BALAKRISHNAN, G., AND REPS, T. Divine: discovering variables in executables. In VMCAI’07 (2007), pp. 1–28.

[5] C. Hawblitzel, S. K. Lahiri, K. Pawar, H. Hashmi, S. Gokbulut, L. Fernando, D. Detlefs, and S.Wadsworth. Will you still compile me tomorrow? static cross-version compiler validation. In Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE’13, Saint Petersburg, Russian Federation, August 18-26, 2013, pages 191–201, 2013.

[6] Ng, B. H., and Prakash, A. Expose: Discovering potential binary code re-use. In Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual (July 2013), pp. 492–501.

[7] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. In 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015, pages 709–724, 2015.

[8] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken. Data driven equivalence checking. In Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications, OOPSLA ’13, pages 391–406, New York, NY, USA, 2013. ACM.

[9] SAEBJORNSEN, A., WILLCOCK, J., PANAS, T., QUINLAN, D., AND SU, Z. Detecting code clones in binary executables. In ISSTA ’09.

[10] Y. David, N. Partush, and E. Yahav. Statistical similarity of binaries. In Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’16, 2016.

[11] HORWITZ, S., REPS, T., AND BINKLEY, D. Interprocedural slicing using dependence graphs. In PLDI ’88 (1988).

[12] BELLON, S., KOSCHKE, R., ANTONIOL, G., KRINKE, J., AND MERLO, E. Comparison and evaluation of clone detection tools. IEEE TSE 33, 9 (2007), 577–591.

[13] BRUSCHI, D., MARTIGNONI, L., AND MONGA, M. Detecting selfmutating malware using control-flow graph matching. In DIMVA’06.

[14] KRUEGEL, C., KIRDA, E., MUTZ, D., ROBERTSON, W., AND VIGNA, G. Polymorphic worm detection using structural information of executables. In Proc. of int. conf. on Recent Advances in Intrusion Detection, RAID’05.

[15] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin. Scalable graph-based bug search for firmware images. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24- 28, 2016, pages 480–491, 2016.

[16] ROSENBLUM, N. E., MILLER, B. P., AND ZHU, X. Extracting compiler provenance from program binaries. In PASTE’10.

[17] JANG, J., WOO, M., AND BRUMLEY, D. Towards automatic software lineage inference. In USENIX Security (2013).

[18] ROSENBLUM, N., ZHU, X., AND MILLER, B. P. Who wrote this code? identifying the authors of program binaries. In ESORICS’11.

[19] Lahiri, S. K., Sinha, R., and Hawblitzel, C. Automatic rootcausing for program equivalence failures in binaries. In Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (2015), pp. 362–379.

[20] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. Symdiff: A language-agnostic semantic di\_ tool for imperative programs. In CAV, pages 712–717, 2012.

[21] SCHKUFZA, E., SHARMA, R., AND AIKEN, A. Stochastic super optimization. In ASPLOS ’13.

[22] Aiken, A. Moss. <https://theory.stanford.edu/~aiken/moss/>.

[23] Smith, R., and Horwitz, S. Detecting and measuring similarity in code clones. In Proceedings of the International Workshop on Software Clones (IWSC) (2009).

[24] N. Partush and E. Yahav. Abstract semantic differencing for numerical programs. In Static Analysis: 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings, pages 238–258. Springer, 2013.

[25] N. Partush and E. Yahav. Abstract semantic differencing via speculative correlation. In Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014, pages 811–828, 2014.

[26] Pewny, J., Schuster, F., Bernhard, L., Holz, T., and Rossow, C. Leveraging semantic signatures for bug search in binary programs In Proceedings of the 30th Annual Computer Security Applications Conference (2014), ACSAC ’14, ACM, pp. 406–415.

[27] Jacobson, E. R., Rosenblum, N., and Miller, B. P. Labeling library functions in stripped binaries In Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools (2011), PASTE ’11, ACM, pp. 1–8.

[28] D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In Proceedings of the 23rd International Conference on Computer Aided Verification, CAV’11, pages 669–685, Berlin, Heidelberg, 2011. Springer-Verlag.

# Appendix

Matlab Code

* Finding C and Weight Vector on random method

filePath = 'C:\Users\User\Desktop\Final Project\Statistics\statistic.mat';

data = load(filePath);

statistic = data.statistic;

% x = number of optimise

% y = number of optimise

% z = number of features

[x, y, z] = size(statistic);

V = zeros(x, z);

for i=1:x

for k=1:z

Vi = 0;

for j=1:y

Vi = Vi + statistic(i, j, k);

end

p = statistic(i, i, k) - (Vi - statistic(i, i, k) );

V(i, k) = p;

end

end

C\_min = -9999999999999999;

W\_save = zeros(1, z);

for r=1:100000000000000000000

W = rand\_norm\_array(z);

S= W\*V';

C = min(S);

if C > C\_min

C\_min = C;

W\_save = W;

disp("found new max C = " + C);

end

end

* Finding C and Weight Vector on optimization method

clear all;

filePath = 'C:\Users\User\Desktop\Final Project\Statistics\statistic.mat';

data = load(filePath);

statistic = data.statistic;

% x = number of optimise

% y = number of optimise

% z = number of features

[x, y, z] = size(statistic);

z = z - 3;

S = zeros(x, y, z);

V = zeros(x, z);

%% Build V vector

for i=1:x % for each function

for k=1:z

ki = statistic(i, 1:y, k);

V(i, k) = 4 \* ki(i) -2 \* sum(ki) + z \* sum(ki);

end

end

%% Doing optimization

fun = @(W)-1 \* V \* W;

options = optimoptions('fminimax', 'AbsoluteMaxObjectiveCount', 1, 'ConstraintTolerance', 5.3663e-19 );

w0 = zeros(z, 1);

ub = ones(z, 1); % x <= 1

lb = zeros(z, 1); % x >= 0

Aeq = ones(1, z); % sum(x) == 1 constraint

beq = 1;

A = -V;

b = zeros(y, 1);

nonlcon = []; % @opt\_func

W = fminimax(fun, w0, A, b, Aeq, beq, lb, ub, nonlcon, options);

%% Showing the result

if sum(W) ~= 1

disp("Sum of W not equal to 1(" + sum(W) + ")")

end

C = min(V \* W);

disp("Found C: " + C)

disp("Found W: " + W);

%% Testing the result

miss = 0;

hit = 0;

for i=1:x

Pi\_ = statistic(i, 1:y, 1:z);

Xi = zeros(1, y);

Pi = reshape(Pi\_, y, z);

Xi\_ = Pi \* W;

sum\_Xi\_ = sum(Xi\_);

for j=1:y

Xi(j) = Xi\_(j) - (sum\_Xi\_ - Xi\_(j));

end

if max(Xi) > Xi(i)

% disp("Problematic function in row #" + i + ", The max is: " + max(Xi) + " and the Xi(i) is: " + Xi(i));

miss = miss + 1;

else

% disp("Function " + i + " is ok, max(Xi) = " + max(Xi) + " min(Xi) = " + min(Xi));

hit = hit + 1;

end

end

disp("==== Total Statistics ====");

disp("AFR algorithm success rates: " + hit + " / " + x + " = " + hit/x + " %");

Python Code

**from** idautils **import** \*  
**from** idaapi **import** \*  
**from** idc **import** \*  
**import** json  
**import** threading  
**import** scipy.io  
**import** idautils  
**import** time  
**import** os  
  
*# Tested on IDA 6.8 Python x64 2.7  
  
# =========== Configurations ===========*release\_library = **'I:\\Final Project\\Libraries\\uclibc\_release\\libuClibc-1.0.28.so'**debug\_library = **'I:\\Final Project\\Libraries\\uclibc\_debug\\libuClibc-1.0.28.so'**base = **'C:\/Users\User\/Desktop\/Final Project\/Statistics\/'** *# 'I:\/Final Project\/Statistics\/'*function\_min\_size = 10  
  
*# =========== global vars ===========*file\_index = 0  
max\_enum = 0  
max\_feature\_enum = 0  
threadLock\_logfile = threading.Lock()  
threadLock\_grades = threading.Lock()  
index = 0  
enum\_map = {}  
enum\_features\_map = {}  
features = {  
 **'Nested function'**, **'Jump match'**, **'Constant match'**,  
 **'Command rare match'**, **'Mnemonic match'**, **'Mnemonic subsequence match'**,  
 **'Amount of menemonic match'**, **'Number of args match'** }  
  
  
*# =========== Utilities ===========***def** json\_read(file):  
 **with** open(file) **as** json\_file:  
 data = json.load(json\_file)  
 **return** data  
  
  
**def** json\_write(file, data):  
 **with** open(file, **'w'**) **as** json\_file:  
 json.dump(data, json\_file)  
  
  
**def** json\_append(file, data):  
 threadLock\_grades.acquire()  
 **try**:  
 **if not** os.path.isfile(file):  
 feeds = {}  
 **else**:  
 feedsjson = open(file, **"r"**)  
 feeds = json.load(feedsjson)  
 feedsjson.flush()  
 feedsjson.close()  
  
 feeds.update(data)  
 f = open(file, **'w'**)  
 f.write(json.dumps(feeds, indent=2))  
 f.flush()  
 f.close()  
 **except** Exception **as** e:  
 **print "Exeption: "** + e  
 threadLock\_grades.release()  
  
  
**def** get\_ea\_start\_func(adderess):  
 func = get\_func(adderess)  
 **return** func.startEA  
  
  
**def** log(file, data):  
 **with** open(file, **"a"**) **as** log\_file:  
 log\_file.write(data)  
  
  
*# =========== Mnemonic match ===========***def** get\_func\_mnemonic\_list(i\_start\_func):  
 mnemonics\_container = []  
  
 func\_instructions = FuncItems(i\_start\_func)  
  
 **for** instruction **in** func\_instructions:  
 menomonic = GetMnem(instruction)  
 **if** menomonic **not in** mnemonics\_container:  
 mnemonics\_container += [menomonic]  
  
 **return** mnemonics\_container  
  
  
**def** mnemonic\_match\_feature(i\_start\_debug\_func, i\_start\_release\_func):  
 debug\_mnemonics\_list = get\_func\_mnemonic\_list(i\_start\_debug\_func)  
 release\_mnemonics\_list = get\_func\_mnemonic\_list(i\_start\_release\_func)  
  
 amount\_of\_containment = 0  
  
 **for** debug\_mnemonics **in** debug\_mnemonics\_list:  
 **if** debug\_mnemonics **in** release\_mnemonics\_list:  
 amount\_of\_containment += 1  
  
 **return** amount\_of\_containment / float(len(release\_mnemonics\_list))  
  
  
**def** mnemonic\_match\_feaure\_helper(i\_debug\_mnemonics\_list, i\_release\_mnemonics\_list):  
 amount\_of\_containment = 0  
  
 **for** debug\_mnemonics **in** i\_debug\_mnemonics\_list:  
 **if** debug\_mnemonics **in** i\_release\_mnemonics\_list:  
 amount\_of\_containment += 1  
  
 **if** len(i\_release\_mnemonics\_list) == 0 **and** amount\_of\_containment == 0:  
 **return** 1  
 **if** len(i\_release\_mnemonics\_list) == 0:  
 **return** 0  
  
 **return** amount\_of\_containment / float(len(i\_release\_mnemonics\_list))  
  
  
*# =========== Command rare match ===========***def** get\_rare\_mnemonic\_list(i\_mnemonics\_dictionary):  
 rare\_mnemonic\_list = []  
  
 **for** mnemonic **in** i\_mnemonics\_dictionary.keys():  
 **if** i\_mnemonics\_dictionary[mnemonic] == 1:  
 rare\_mnemonic\_list += [mnemonic]  
  
 **return** rare\_mnemonic\_list  
  
  
**def** get\_mnemonic\_dictionary(i\_start\_func):  
 func\_instructions = FuncItems(i\_start\_func)  
 mnemonics\_dictionary = {}  
  
 **for** instruction **in** func\_instructions:  
 mnemonic = GetMnem(instruction)  
 **if** mnemonic **not in** mnemonics\_dictionary:  
 mnemonics\_dictionary[mnemonic] = 1  
 **else**:  
 mnemonics\_dictionary[mnemonic] += 1  
  
 **return** mnemonics\_dictionary  
  
  
**def** get\_rare\_instruction\_list(i\_start\_func):  
 rare\_commands\_list = []  
  
 mnemonic\_dictionary = get\_mnemonic\_dictionary(i\_start\_func)  
 rare\_mnemonic\_list = get\_rare\_mnemonic\_list(mnemonic\_dictionary)  
  
 func\_instructions = FuncItems(i\_start\_func)  
  
 **for** instruction **in** func\_instructions:  
 mnemonic = GetMnem(instruction)  
 **if** mnemonic **in** rare\_mnemonic\_list:  
 disasm = GetDisasm(instruction).split(**";"**)  
 rare\_commands\_list += [disasm[0]]  
  
 **return** rare\_commands\_list  
  
  
**def** get\_instructions\_list(i\_start\_func):  
 commands\_list = []  
  
 func\_instructions = FuncItems(i\_start\_func)  
 **for** instruction **in** func\_instructions:  
 instruction = GetDisasm(instruction).split(**";"**)  
 commands\_list += [instruction[0]]  
  
 **return** commands\_list  
  
  
**def** rare\_commands\_feature(i\_start\_debug\_func, i\_start\_release\_func):  
 rare\_release\_instruction\_list = get\_rare\_instruction\_list(i\_start\_release\_func)  
 debug\_instruction\_list = get\_instructions\_list(i\_start\_debug\_func)  
  
 rare\_command\_in\_release\_exist\_in\_debug = 0  
  
 **for** release\_command **in** rare\_release\_instruction\_list:  
 **if** release\_command **in** debug\_instruction\_list:  
 rare\_command\_in\_release\_exist\_in\_debug += 1  
  
 **return** rare\_command\_in\_release\_exist\_in\_debug / float(len(rare\_release\_instruction\_list))  
  
  
**def** rare\_commands\_feature\_helper(i\_debug\_instruction\_list, i\_rare\_release\_instruction\_list):  
 rare\_command\_in\_release\_exist\_in\_debug = 0  
  
 **for** release\_command **in** i\_rare\_release\_instruction\_list:  
 **if** release\_command **in** i\_debug\_instruction\_list:  
 rare\_command\_in\_release\_exist\_in\_debug += 1  
  
 **if** len(i\_rare\_release\_instruction\_list) == 0 **and** rare\_command\_in\_release\_exist\_in\_debug == 0:  
 **return** 1  
 **if** len(i\_rare\_release\_instruction\_list) == 0:  
 **return** 0  
 **return** rare\_command\_in\_release\_exist\_in\_debug / float(len(i\_rare\_release\_instruction\_list))  
  
  
*# =========== Constant match ===========* ***TODO: to take the 0.5 grade, and dropping jump instructions*def** get\_constant\_list(i\_start\_func):  
 constant\_list = []  
 func\_instructions = FuncItems(i\_start\_func)  
  
 **for** instruction **in** func\_instructions:  
 operand\_type\_0 = GetOpType(instruction, 0)  
 operand\_type\_1 = GetOpType(instruction, 1)  
 operand\_type\_2 = GetOpType(instruction, 2)  
  
 **if** operand\_type\_0 == o\_imm **or** operand\_type\_0 == o\_mem:  
 op = GetOpnd(instruction, 0)  
 **if** op **not in** constant\_list:  
 constant\_list += [op]  
  
 **if** operand\_type\_1 == o\_imm **or** operand\_type\_1 == o\_mem:  
 op = GetOpnd(instruction, 1)  
 **if** op **not in** constant\_list:  
 constant\_list += [op]  
  
 **if** operand\_type\_2 == o\_imm **or** operand\_type\_2 == o\_mem:  
 op = GetOpnd(instruction, 2)  
 **if** op **not in** constant\_list:  
 constant\_list += [op]  
  
 **return** constant\_list  
  
  
**def** constant\_match\_feature(i\_start\_debug\_func, i\_start\_release\_func):  
 release\_constant\_list = get\_constant\_list(i\_start\_debug\_func)  
 debug\_constant\_list = get\_constant\_list(i\_start\_release\_func)  
  
 amount\_of\_release\_function\_in\_debug = 0  
  
 **for** constant **in** release\_constant\_list:  
 **if** constant **in** debug\_constant\_list:  
 amount\_of\_release\_function\_in\_debug += 1  
  
 **return** amount\_of\_release\_function\_in\_debug / float(len(release\_constant\_list))  
  
  
**def** constant\_match\_feature\_helper(i\_debug\_constant\_list, i\_release\_constant\_list):  
 amount\_of\_release\_function\_in\_debug = 0  
  
 **for** constant **in** i\_release\_constant\_list:  
 **if** constant **in** i\_debug\_constant\_list:  
 amount\_of\_release\_function\_in\_debug += 1  
  
 **if** len(i\_release\_constant\_list) == 0 **and** amount\_of\_release\_function\_in\_debug == 0:  
 **return** 1  
 **if** len(i\_release\_constant\_list) == 0:  
 **return** 0  
 **return** amount\_of\_release\_function\_in\_debug / float(len(i\_release\_constant\_list))  
  
  
*# =========== Jump match ===========***def** get\_jump\_list(i\_start\_func):  
 jump\_list = []  
  
 func\_instructions = FuncItems(i\_start\_func)  
  
 **for** instruction **in** func\_instructions:  
 mnemonic = GetMnem(instruction)  
 **if** mnemonic **in** [**'jmp'**, **'ja'**, **'jae'**, **'jb'**, **'jbe'**, **'je'**, **'jg'**, **'jge'**, **'jlr'**, **'jna'**, \  
 **'jnae'**, **'jnb'**, **'jnbe'**, **'jc'**, **'jcxz'**, **'jecxz'**, **'jnc'**, **'jne'**, **'jng'**, \  
 **'jnge'**, **'jnl'**, **'jnle'**, **'jno'**, **'jnp'**, **'jns'**, **'jnz'**, **'jo'**, **'jp'**, **'jpe'**, \  
 **'jpo'**, **'js'**, **'jz'**, **'ja'**, **'jae'**, **'jb'**, **'jbe'**, **'jc'**, **'je'**, **'jz'**, **'jg'**, \  
 **'jle'**, **'jl'**]:  
 jump\_list += [mnemonic]  
  
 **return** jump\_list  
  
  
**def** jump\_match\_feature(i\_start\_debug\_func, i\_start\_release\_func):  
 release\_jump\_list = get\_jump\_list(i\_start\_debug\_func)  
 debug\_jump\_list = get\_jump\_list(i\_start\_release\_func)  
  
 **return** len(release\_jump\_list) / float(len(debug\_jump\_list))  
  
  
**def** jump\_match\_feature\_helper(i\_debug\_jump\_list, i\_release\_jump\_list):  
 **if** len(i\_debug\_jump\_list) == 0 **and** len(i\_release\_jump\_list) == 0:  
 **return** 1  
 **if** len(i\_debug\_jump\_list) == 0:  
 **return** 0  
  
 result = len(i\_release\_jump\_list) / float(len(i\_debug\_jump\_list))  
  
 **if** result > 1.0:  
 **return** 1  
 **else**:  
 **return** result  
  
  
*# =========== Nested function match ===========***def** get\_func\_name\_list(i\_start\_func):  
 nested\_func\_list = []  
  
 func\_instructions = FuncItems(i\_start\_func)  
  
 **for** instruction **in** func\_instructions:  
 mnemonic = GetMnem(instruction)  
 **if** mnemonic **in** [**'call'**]:  
 function = GetOpnd(instruction, 0)  
 nested\_func\_list += [function]  
 **return** nested\_func\_list  
  
  
**def** nested\_function\_match\_feature(i\_start\_debug\_func, i\_start\_release\_func):  
 release\_func\_name\_list = get\_func\_name\_list(i\_start\_debug\_func)  
 debug\_func\_name\_list = get\_func\_name\_list(i\_start\_release\_func)  
  
 amount\_of\_release\_and\_debug = 0  
  
 **for** function\_name **in** release\_func\_name\_list:  
 **if** function\_name **in** debug\_func\_name\_list:  
 amount\_of\_release\_and\_debug += 1  
  
 **return** amount\_of\_release\_and\_debug / float(len(debug\_func\_name\_list))  
  
  
**def** nested\_function\_match\_feature\_helper(i\_debug\_func\_name\_list, i\_release\_func\_name\_list):  
 amount\_of\_release\_and\_debug = 0  
 checked\_functions = []  
  
 **for** debug\_func\_name **in** i\_debug\_func\_name\_list:  
 **for** release\_func\_name **in** i\_release\_func\_name\_list:  
 pos = debug\_func\_name.find(release\_func\_name)  
 **if** -1 != pos **and** release\_func\_name **not in** checked\_functions:  
 checked\_functions += [release\_func\_name]  
 amount\_of\_release\_and\_debug += 1  
  
 **if** len(i\_release\_func\_name\_list) == 0 **and** amount\_of\_release\_and\_debug == 0:  
 **return** 1  
 **if** len(i\_release\_func\_name\_list) == 0:  
 **return** 0  
  
 result = amount\_of\_release\_and\_debug / float(len(i\_release\_func\_name\_list))  
  
 **if** result > 1.0:  
 **return** 1  
 **else**:  
 **return** result  
  
  
*# =========== Number of args match ===========***def** get\_number\_of\_args(i\_start\_address):  
 tif = tinfo\_t()  
 **if not** get\_tinfo2(i\_start\_address, tif):  
 guess\_tinfo2(i\_start\_address, tif)  
 function\_data = func\_type\_data\_t()  
 tif.get\_func\_details(function\_data)  
 **return** function\_data.size()  
  
  
**def** number\_of\_args\_match\_feature(i\_start\_debug\_func, i\_start\_release\_func):  
 release\_func\_number\_of\_args = get\_number\_of\_args(i\_start\_debug\_func)  
 debug\_func\_number\_of\_args = get\_number\_of\_args(i\_start\_release\_func)  
  
 **if** release\_func\_number\_of\_args == debug\_func\_number\_of\_args:  
 **return** 1.0  
 **return** 0.0  
  
  
**def** number\_of\_args\_match\_helper(i\_debug\_func\_number\_of\_args, i\_release\_func\_number\_of\_args):  
 **if** i\_release\_func\_number\_of\_args == i\_debug\_func\_number\_of\_args:  
 **return** 1.0  
 **return** 0.0  
  
  
*# =========== amount of menemonic match ===========***def** get\_amount\_of\_mnemonic(i\_start\_address):  
 func\_instructions = FuncItems(i\_start\_address)  
 **return** len(list(func\_instructions))  
  
  
**def** amount\_of\_menemonic\_match\_feature(i\_start\_debug\_func, i\_start\_release\_func):  
 release\_func\_amount\_of\_mnemonic = get\_amount\_of\_mnemonic(i\_start\_debug\_func)  
 debug\_func\_amount\_of\_mnemonic = get\_amount\_of\_mnemonic(i\_start\_release\_func)  
  
 **if** release\_func\_amount\_of\_mnemonic == debug\_func\_amount\_of\_mnemonic:  
 **return** 1.0  
 **return** 0.0  
  
  
**def** amount\_of\_mnemonic\_match\_helper(debug\_func\_amount\_of\_mnemonic, release\_func\_amount\_of\_mnemonic):  
 **if** debug\_func\_amount\_of\_mnemonic == 0:  
 **return** 0.0  
  
 **if** release\_func\_amount\_of\_mnemonic >= debug\_func\_amount\_of\_mnemonic:  
 **return** 1.0  
  
 **return** release\_func\_amount\_of\_mnemonic / float(debug\_func\_amount\_of\_mnemonic)  
  
  
*# =========== longest Common menomonic subsequence of menemonic match ===========***def** lcs(X, Y):  
 *# find the length of the strings* m = len(X)  
 n = len(Y)  
  
 *# declaring the array for storing the dp values* L = [[None] \* (n + 1) **for** i **in** xrange(m + 1)]  
  
 **"""Following steps build L[m+1][n+1] in bottom up fashion   
 Note: L[i][j] contains length of LCS of X[0..i-1]   
 and Y[0..j-1]"""  
 for** i **in** range(m + 1):  
 **for** j **in** range(n + 1):  
 **if** i == 0 **or** j == 0:  
 L[i][j] = 0  
 **elif** X[i - 1] == Y[j - 1]:  
 L[i][j] = L[i - 1][j - 1] + 1  
 **else**:  
 L[i][j] = max(L[i - 1][j], L[i][j - 1])  
  
 *# L[m][n] contains the length of LCS of X[0..n-1] & Y[0..m-1]* **return** L[m][n]  
  
  
**def** get\_sequence(i\_start\_func):  
 sequence = **""** func\_instructions = FuncItems(i\_start\_func)  
  
 **for** instruction **in** func\_instructions:  
 mnemonic = GetMnem(instruction)  
 sequence += mnemonic  
  
 **return** sequence  
  
  
**def** mnemonic\_subsequence\_match\_feature(i\_start\_debug\_func, i\_start\_release\_func):  
 release\_func\_sequence = get\_sequence(i\_start\_debug\_func)  
 debug\_func\_sequence = get\_sequence(i\_start\_release\_func)  
  
 **if** len(debug\_func\_sequence) == 0 **or** len(release\_func\_sequence):  
 **return** 0  
 lcs\_sequence = lcs(debug\_func\_sequence, release\_func\_sequence)  
  
 **return** lcs\_sequence / float(len(release\_func\_sequence))  
  
  
**def** mnemonic\_subsequence\_match\_helper(i\_debug\_func\_sequence, i\_release\_func\_sequence):  
 **if** len(i\_debug\_func\_sequence) == 0 **or** len(i\_release\_func\_sequence) == 0:  
 **return** 0  
 lcs\_sequence = lcs(i\_debug\_func\_sequence, i\_release\_func\_sequence)  
  
 **return** lcs\_sequence / float(len(i\_release\_func\_sequence))  
  
  
*# =============================================  
# ================ statistics ================  
# =============================================***def** get\_export\_func\_name(address):  
 **for** func **in** list(Entries()):  
 **if** address == func[1]:  
 **return** func[3]  
 **return** None  
  
  
**def** get\_function\_size(address):  
 **return** len(list(idautils.FuncItems(address)))  
  
  
**def** get\_all\_library\_grades(functions):  
 function\_map = {}  
 **for** f **in** functions:  
 func\_name = get\_export\_func\_name(f)  
 **if** func\_name == None:  
 func\_name = get\_func\_name(f)  
  
 features\_map = {}  
  
 features\_map[**'Nested function'**] = get\_func\_name\_list(f)  
 features\_map[**'Jump match'**] = get\_jump\_list(f)  
 features\_map[**'Constant match'**] = get\_constant\_list(f)  
 features\_map[**'Command rare match'**] = get\_rare\_instruction\_list(f)  
 features\_map[**'Mnemonic match'**] = get\_func\_mnemonic\_list(f)  
 features\_map[**'Mnemonic subsequence match'**] = get\_sequence(f)  
 features\_map[**'Amount of menemonic match'**] = get\_amount\_of\_mnemonic(f)  
 features\_map[**'Number of args match'**] = get\_number\_of\_args(f)  
  
 function\_map[func\_name] = features\_map  
 **return** function\_map  
  
  
**def** get\_export\_functions\_map():  
 export\_functions = []  
 **for** func **in** list(Entries()):  
 **if** function\_min\_size < get\_function\_size(func[2]):  
 export\_functions += [[func[2], func[3]]]  
 **return** export\_functions  
  
  
**def** get\_export\_functions():  
 export\_functions = []  
 **for** func **in** list(Entries()):  
 **if** function\_min\_size < get\_function\_size(func[2]):  
 export\_functions += [func[2]]  
 **return** export\_functions  
  
  
**def** get\_func\_ea\_list(i\_start\_func):  
 nested\_func\_list = []  
 func\_instructions = FuncItems(i\_start\_func)  
  
 **for** instruction **in** func\_instructions:  
 mnemonic = GetMnem(instruction)  
 **if** mnemonic **in** [**'call'**]:  
 function\_ea = GetOperandValue(instruction, 0)  
 **if** function\_ea != 0:  
 nested\_func\_list += [function\_ea]  
 **return** nested\_func\_list  
  
  
**def** build\_func\_graph(start\_func, visited\_functions):  
 func\_graph = {}  
 visited\_functions += [start\_func]  
 nested\_func\_name\_list = get\_func\_ea\_list(start\_func)  
 **for** func **in** nested\_func\_name\_list:  
 **if** func **not in** visited\_functions:  
 func\_name = get\_export\_func\_name(func)  
 **if** func\_name == None:  
 func\_name = get\_func\_name(func)  
 func\_graph[func\_name] = build\_func\_graph(func, visited\_functions)  
 **return** func\_graph  
  
  
**def** build\_graph(export\_functions):  
 all\_export\_func\_graph = {}  
 **for** f **in** export\_functions:  
 func\_address = f[0]  
 func\_name = f[1]  
 func\_graph = build\_func\_graph(func\_address, [])  
 all\_export\_func\_graph[func\_name] = func\_graph  
 **return** all\_export\_func\_graph  
  
  
**def** thread\_job(func\_name\_debug, index):  
 final\_grades = {}  
 final\_debug\_grades = {}  
  
 debug\_function\_grades = debug\_library\_grades[func\_name\_debug]  
  
 debug\_function\_nested\_function\_grade = debug\_function\_grades[**'Nested function'**]  
 debug\_function\_jump\_match\_grade = debug\_function\_grades[**'Jump match'**]  
 debug\_function\_constant\_match\_grade = debug\_function\_grades[**'Constant match'**]  
 debug\_function\_command\_rare\_match\_grade = debug\_function\_grades[**'Command rare match'**]  
 debug\_function\_mnemonic\_match\_grade = debug\_function\_grades[**'Mnemonic match'**]  
 debug\_function\_mnemonic\_subsequence\_match\_grade = debug\_function\_grades[**'Mnemonic subsequence match'**]  
 debug\_function\_amount\_of\_menemonic\_match\_grade = debug\_function\_grades[**'Amount of menemonic match'**]  
 debug\_function\_number\_of\_args\_match\_grade = debug\_function\_grades[**'Number of args match'**]  
  
 **for** func\_name\_release **in** functions\_name\_array:  
 features\_map = {}  
  
 release\_grades = release\_library\_grades[func\_name\_release]  
  
 release\_function\_nested\_function\_grade = release\_grades[**'Nested function'**]  
 release\_function\_jump\_match\_grade = release\_grades[**'Jump match'**]  
 release\_function\_constant\_match\_grade = release\_grades[**'Constant match'**]  
 release\_function\_command\_rare\_match\_grade = release\_grades[**'Command rare match'**]  
 release\_function\_mnemonic\_match\_grade = release\_grades[**'Mnemonic match'**]  
 release\_function\_mnemonic\_subsequence\_match\_grade = release\_grades[**'Mnemonic subsequence match'**]  
 release\_function\_amount\_of\_menemonic\_match\_grade = release\_grades[**'Amount of menemonic match'**]  
 release\_function\_number\_of\_args\_match\_grade = release\_grades[**'Number of args match'**]  
  
 features\_map[**'Nested function'**] = nested\_function\_match\_feature\_helper(debug\_function\_nested\_function\_grade,  
 release\_function\_nested\_function\_grade)  
 features\_map[**'Jump match'**] = jump\_match\_feature\_helper(debug\_function\_jump\_match\_grade,  
 release\_function\_jump\_match\_grade)  
 features\_map[**'Constant match'**] = constant\_match\_feature\_helper(debug\_function\_constant\_match\_grade,  
 release\_function\_constant\_match\_grade)  
 features\_map[**'Command rare match'**] = rare\_commands\_feature\_helper(debug\_function\_command\_rare\_match\_grade,  
 release\_function\_command\_rare\_match\_grade)  
 features\_map[**'Mnemonic match'**] = mnemonic\_match\_feaure\_helper(debug\_function\_mnemonic\_match\_grade,  
 release\_function\_mnemonic\_match\_grade)  
 features\_map[**'Mnemonic subsequence match'**] = mnemonic\_subsequence\_match\_helper(  
 debug\_function\_mnemonic\_subsequence\_match\_grade, release\_function\_mnemonic\_subsequence\_match\_grade)  
 features\_map[**'Amount of menemonic match'**] = amount\_of\_mnemonic\_match\_helper(  
 debug\_function\_amount\_of\_menemonic\_match\_grade, release\_function\_amount\_of\_menemonic\_match\_grade)  
 features\_map[**'Number of args match'**] = number\_of\_args\_match\_helper(debug\_function\_number\_of\_args\_match\_grade,  
 release\_function\_number\_of\_args\_match\_grade)  
  
 final\_debug\_grades[func\_name\_release] = features\_map  
  
 final\_grades[func\_name\_debug] = final\_debug\_grades  
 **print "flashing to file: "**, index, **" thread name: "**, threading.current\_thread().name  
  
 json\_append(base + **'final\_grades\_n\_by\_n\_'** + str(file\_index) + **'.txt'**, final\_grades)  
  
  
**def** get\_enum\_by\_name(i\_enum\_map, i\_name):  
 **return** i\_enum\_map[i\_name]  
  
  
**def** get\_name\_by\_enum(i\_enum\_map, i\_enum):  
 keys = i\_enum\_map.keys()  
 **for** key **in** keys:  
 **if** i\_enum\_map[key] == i\_enum:  
 **return** key  
 **return** 0  
  
  
**def** set\_enum(i\_enum\_map, i\_name):  
 i\_enum\_map[i\_name] = get\_enum()  
  
  
**def** get\_enum():  
 **global** max\_enum  
 max\_enum = max\_enum + 1  
 **return** max\_enum - 1  
  
  
**def** get\_feature\_enum():  
 **global** max\_feature\_enum  
 max\_feature\_enum = max\_feature\_enum + 1  
 **return** max\_feature\_enum - 1  
  
  
**def** set\_feature\_enum(i\_enum\_features\_map, i\_name):  
 i\_enum\_features\_map[i\_name] = get\_feature\_enum()  
  
  
**def** get\_enum\_by\_feature(i\_enum\_features\_map, i\_name):  
 **return** i\_enum\_features\_map[i\_name]  
  
  
*# ==================================================================  
# ============================== main ==============================  
# ==================================================================  
  
  
# 1. Build grades for debug library*export\_functions = get\_export\_functions()  
debug\_library\_grades = get\_all\_library\_grades(export\_functions)  
  
json\_write(base + **'uclibc\_debug\_function\_map.txt'**, debug\_library\_grades)  
  
*# 2. Build grades for debug graph*export\_functions = get\_export\_functions\_map()  
debug\_library\_export\_graph = build\_graph(export\_functions)  
  
json\_write(base + **'uclibc\_debug\_export\_function\_map.txt'**, debug\_library\_export\_graph)  
  
*# 3. Build release function graph*export\_functions = get\_export\_functions()  
release\_library\_grades = get\_all\_library\_grades(export\_functions)  
  
json\_write(base + **'uclibc\_release\_function\_map.txt'**, release\_library\_grades)  
  
*# 4. grades for release function library*export\_functions = get\_export\_functions\_map()  
release\_library\_export\_graph = build\_graph(export\_functions)  
  
json\_write(base + **'uclibc\_release\_export\_function\_map.txt'**, release\_library\_export\_graph)  
  
*# 5. Merge grades (From this phase can running without IDA )*debug\_library\_grades = json\_read(base + **'uclibc\_debug\_function\_map.txt'**)  
release\_library\_grades = json\_read(base + **'uclibc\_release\_function\_map.txt'**)  
debug\_library\_export\_graph = json\_read(base + **'uclibc\_debug\_export\_function\_map.txt'**)  
release\_library\_export\_graph = json\_read(base + **'uclibc\_release\_export\_function\_map.txt'**)  
  
 *# release\_library\_grades ['<function name>'] = '<grades>'  
 # debug\_library\_grades ['<function name>'] = '<grades>'  
  
 # debug\_library\_export\_graph ['<function name>'] = '<sub functions>'  
 # release\_library\_export\_graph ['<function name>'] = '<sub functions>'*functions\_name\_array = []  
  
**for** func\_name **in** debug\_library\_export\_graph:  
 **try**:  
 **if** func\_name != **"None"**:  
 release\_func\_tree = release\_library\_export\_graph[func\_name]  
  
 debug\_func\_tree = debug\_library\_export\_graph[func\_name]  
  
 **if** release\_func\_tree != None **and** debug\_func\_tree != None:  
 release\_func\_grades = release\_library\_grades[func\_name]  
 debug\_func\_grades = debug\_library\_grades[func\_name]  
  
 **if** release\_func\_grades != None **and** debug\_func\_grades != None:  
 functions\_name\_array += [func\_name] *# functions\_name\_array[index] = func\_name* **except** KeyError **as** e:  
 **pass  
  
for** func\_name\_debug **in** functions\_name\_array:  
 **while** True:  
 **if** len(threading.enumerate()) < 14:  
 t = threading.Thread(target=thread\_job, args=(func\_name\_debug, index))  
 t.start()  
 index += 1  
 **if** index % 500 == 0:  
 file\_index += 1  
 **break  
 else**:  
 time.sleep(10)  
  
  
  
*# 6. export grades to Matlab*statistic = json\_read(base + **'n\_by\_n\\final\_grades\_n\_by\_n.txt'**)  
statistic\_len = len(statistic)  
features\_len = len(features)  
  
matrix\_scores = [[[0 **for** k **in** xrange(features\_len)]  
 **for** j **in** xrange(statistic\_len)] **for** i **in** xrange(statistic\_len)] *##***for** feature **in** features:  
 set\_feature\_enum(enum\_features\_map, feature)  
  
**for** item **in** statistic:  
 set\_enum(enum\_map, item)  
  
**for** item **in** statistic:  
 row = get\_enum\_by\_name(enum\_map, item)  
 sub\_statistic = statistic[item]  
  
 **for** sub\_item **in** sub\_statistic:  
 col = get\_enum\_by\_name(enum\_map, sub\_item)  
 stat = sub\_statistic[sub\_item]  
  
 **for** feature **in** features:  
 matrix\_scores[row][col][get\_enum\_by\_feature(enum\_features\_map, feature)] = stat[feature]  
  
json\_write(base + **'enum\_map.json'**, enum\_map)  
json\_write(base + **'enum\_features\_map.json'**, enum\_features\_map)  
scipy.io.savemat(base + **'statistic.mat'**, mdict={**"statistic"**: matrix\_scores})